INTERNET PROTOCOL GEOLOCATION: DEVELOPMENT OF A DELAY-BASED HYBRID METHODOLOGY FOR LOCATING THE GEOGRAPHIC LOCATION OF A NETWORK NODE

THESIS

John M. Roehl, Captain, USAF

March 2007

AFIT/GIR/ENV/07-M15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

“The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.”

AFIT/GIR/ENV/07-M15

INTERNET PROTOCOL GEOLOCATION: DEVELOPMENT OF A DELAY-BASED HYBRID METHODOLOGY FOR GEOGRAPHIC LOCATION OF A NETWORK NODE

THESIS

Presented to the Faculty

Department of Systems and Engineering Management

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Information Resource Management

John M. Roehl, BA

Captain, USAF

March 2007

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GIR/ENV/07-M15

INTERNET PROTOCOL GEOLOCATION: DEVELOPMENT OF A DELAY-BASED HYBRID METHODOLOGY FOR GEOGRAPHIC LOCATION OF A NETWORK NODE

John M. Roehl, BA
Captain, USAF

Approved:

// SIGNED //	16 Mar 07
Michael R. Grimaila, PhD, CISM, CISSP, GSEC (Chairman)	Date
// SIGNED //	16 Mar 07
Maj. Jason M. Turner, PhD (Member)	Date
// SIGNED //	16 Mar 07
Maj. Frederick G. Harmon, PhD (Member)	Date

Abstract

Internet Protocol Geolocation (IP Geolocation), the process of determining the approximate geographic location of an IP addressable node, has proven useful in a wide variety of commercial applications. Commercial applications of IP Geolocation include market research, redirection for performance enhancement, restricting content, and combating fraud. The potential for military applications include securing remote access via geographic authentication, intelligence collection, and cyber attack attribution.

IP Geolocation methods can be divided into three basic categories based upon what information is used to determine the geographic location of the given IP address: 1) Information contained in databases, 2) information that is leaked during connections with the IP of interest, and 3) network-based routing and timing information. This thesis focuses upon an analysis in the third category: delay-based methods. Specifically, a comparative analysis of the three existing delay-based IP Geolocation methods: Upper-bound Multilateration (UBM), Constraint Based Geolocation (CBG), and Time to Location Heuristic (TTLH) is conducted. Based upon analysis of the results, a new hybrid methodology is proposed that combines the three existing methods to improve the accuracy when conducting IP Geolocation. Simulations results showed that the new hybrid methodology TTLH method improved the success rate from 80.15% to 91.66% when compared to the shotgun TTLH method.

Table of Contents

Page

Abstract iv

Table of Contents v

List of Figures vi

List of Tables vii

1. Introduction 1

1.1 Problem Statement 4
1.2 Research Objectives 4
1.3 Research Methodology 5
1.4 Research Significance 6
1.5 Thesis Overview 6

2. Literature Review 8

2.1 Chapter Overview 8
2.2 Definition of Terms 8
2.3 Internet Protocol Geolocation 10
2.4 Database Focused Methods 10
2.4.1 WHOIS 11
2.4.2 Domain Name Service (DNS) 12
2.4.3 IP Clustering 12
2.5 Information Leakage-Focused Methods 13
2.5.1 Reverse DNS 14
2.5.2 Trace Route 14
2.5.3 Application Information Leakage 15
2.5.3.1 Finding End-User IP Addresses 16
2.6 Network Communication Attribute-Focused Methods 16
2.6.1 Delay Factors in IP Network Communications 16
2.6.1.2 Topology 17
2.6.1.3 Line Speed 19
2.6.1.4 Queuing Delay 19
2.6.1.5 Switching Speed 20
2.6.1.6 IP Path Diversity 21
2.6.2 Measuring Delay 22
2.6.2.1 Trace Route 23
2.7 Delay-Based IP Geolocation Methods 24

Page

2.7.1 Upper-Bound Multilateration 25
2.7.2 Constraint-Based Geolocation 27
2.7.3 Nearest Known Node 28
2.7.3.1 Euclidean Distance 31
2.8 Summary 34

3. Methodology 36

3.3 Enumerating Network Architectures 39
3.4 Data Collection Process 40
3.5 Data Analysis Process 41
3.5.1 Upper-Bound Multilateration (UBM) 41
3.5.2 Constraint-Based Geolocation (CBG) 42
3.3.3 Time to Location Heuristic (TTLH) 46
3.4 Metrics 48
3.5 Assumptions 53
3.6 Summary 54

4. Results and Analysis 55

4.1 Data Collection 55
4.1 Upper-Bound Multilateration 58
4.1.1 UBM Results 62
4.2 Constraint-Based Geolocation 65
4.2.1 CBG Results 65
4.3 TTLH 72
4.3.1 TTLH Results 78
4.4 Hybrid Methodology 80
4.4.1 Hybrid Methodology Results 81
4.5 Summary 82

5. Conclusions and Recommendations 83

5.1 Summary of Results 84
5.2 Significance of Research 86
5.3 Limitations 87
5.4 Future Research 87

APPENDICIES 89

Appendix A: CollectIt12.c “C” Program 90
Appendix B: CollectIt12.c “C” Program 149

Bibliography 170

List of Figures

Figure Page
1 Delay Triangulation IP Geolocation Method...……..……..……...26
2 Hypothetical Zero-Bit Packet…….…...…...…...30
3 Euclidean Distance Methodology ……..……...….....32
4 Network Topology…….……...……..38
5 Network Architecture Enumeration…….39
6 Boston (P1) Bestline……..……..……...…….44
7 Seattle (P11) Bestline……..……..…...……..48
8 Miss Distance…….49
9 Area of Overlap…….…….…….52
10 UBM results for T2 (Hartford)……...……...…..63
11 Figure 11. San Jose (P10) Bestline …….67
12 CBG results for T2 (Hartford) ……...……..71
13 TTLH Scatterplot of Polling Nodes and Accuracy…….……...……...……73

List of Tables

Table Page
1 Node Designators……....39
2 TTLH Research Design…….…….48
3 Delay Time (seconds)…….……...57
4 Estimated Miles to Targets UBM…….…….59
5 Driving Distance between Nodes…….……...60
6 Number of Miles overestimated by UBM…….…….....……..…….61
7 UBM Results (Area)…….…….……..……...….63
8 UBM Error rates…….…….…….…….……...64
9 Number of Miles Overestimated by CBG…….…….……68
10 CBG Error Rate…….…….…….……69
11 Equation of the Bestlines used in CBG…….…….....70
12 CBG Results (Area)…….……..……...71
13 Summary of CBG and UBM results……..72
14 TTLH Results 2 Polling Nodes and 9 End Nodes……..……...74
15 TTLH Results 3 Polling Nodes and 8 End Nodes……..……...75
16 TTLH Results 4 Polling Nodes and 7 End Nodes……..……...75
17 TTLH Results 5 Polling Nodes and 6 End Nodes……..……...76
18 TTLH Results 6 Polling Nodes and 5 End Nodes……..……...76
19 TTLH Results 7 Polling Nodes and 4 End Nodes……..……...77
20 TTLH Results 8 Polling Nodes and 3 End Nodes……..……...77
21 TTLH Results 9 Polling Nodes and 2 End Nodes……..……...78

INTERNET PROTOCOL GEOLOCATION: DEVELOPMENT OF A DELAY-BASED HYBRID METHODOLOGY FOR LOCATING THE GEOGRAPHIC LOCATION OF A NETWORK NODE

1. Introduction

The evolution of the Internet has revolutionized modern society by providing unprecedented access to information and knowledge to anyone, from anywhere, and at anytime. The systems, infrastructure, and networks that make up the Internet (collectively known as "Cyberspace") transcend all physical boundaries. The Internet provides a means for anyone who has access to an Internet connection to collect or disseminate information on a global scale at a very low cost.

The increasing use and dependence upon cyberspace by public, private, governmental, and military organizations drastically increases our exposure to adversarial activities. Sophisticated hacking tools are freely available on the Internet and enable script-kiddies with few resources and little knowledge to conduct sophisticated information system attacks against individuals and organizations that rival those launched by well-resourced nation states. The purposes of these attacks can range from simple mischief to wide spread attacks against our critical national infrastructure. Our adversaries conduct intelligence collection activities on network connected DOD resources on a daily basis in order to collect, correlate, and exploit information for their nefarious purposes. In all of these cases, it is desirable to identify the ultimate source of

the attack (attribution). Identification of the source requires identification of the physical location of the attacking system (e.g, tracing the attack back and attributing it to a specific IP address), identification of the system(s) which are controlling the attacking system as in many cases the attacking system is an "agent" of the actual attacker, identification of the individuals responsible for initiating the attack, and identification of the organization that is sponsoring the attack. This thesis focuses on methods that address the first element: identifying the physical location of the system.

IP Geolocation is the process of identifying the approximate physical location of a networked device that is connected to the Internet based upon its IP address and communication characteristics. Various techniques for IP Geolocation have appeared in the academic literature [15] [6] [7], some of which have resulted in issuance of US patents, including one by the National Security Agency [16] [8] [1]. IP Geolocation methods can be divided into three basic categories based upon what information is used to determine the geographic location of the given IP address: 1) information contained in databases, 2) information that is leaked during connections with the IP of interest, and 3) network-based routing and timing information. The scope of this thesis is limited to analysis in the third category: delay-based methods for IP Geolocation. The narrow scope of the thesis was intentional and will fill a significant gap in the academic literature. The scoping of the thesis does not suggest that one should only use delay-based IP Geolocation methods. To the contrary, it is recommended that methods from multiple categories should be used to provide as much intelligence as possible. The data can be gained from multiple, non-overlapping, information sources.

IP Geolocation has proven useful in a wide variety of applications. To date, the use of IP Geolocation in published academic literature and applications have been primarily in the commercial arena. For example, commercial organizations are using IP Geolocation for targeted marketing [20]. This provides the ability for web site owners to identify the geographic location of a system accessing their web pages and customize their advertisements by promoting business that are in close proximity to the user. IP Geolocation has been used to combat identity theft. Irregular location patterns from single users would be able to trigger fraud alerts [20]. If a business can identify where a purchase request is coming from, they can act accordingly if the request is from a high fraud area. The result is that organizations can avoid losses and keep costs down. In some cases, knowing the location of the source of a network communication can prevent illegal activities. For example, despite the fact that online gaming web sites are accessible from anywhere on the Internet, gambling is not legal worldwide. Ensuring that users are located in areas where online gaming is legal can keep a business operating lawfully [10]. Law enforcement can use IP Geolocation to locate computer equipment suspected of being used for illegal activities.

While the military use of IP Geolocation is relatively new, it can provide enormous benefits. For example, IP Geolocation provides the ability to help secure remote access via geographic authentication, improve intelligence collection, and enable cyber attack attribution. The military significance of IP Geolocation has dramatically increased with the articulation of Cyberspace as a medium for warfighting within the United States Air Force (USAF) mission statement. On 7 December 2005, Secretary of the Air Force Michael Wynne adopted the new mission statement which states, “The

mission of the United States Air Force (USAF) is to deliver sovereign options for the defense of the United States of America and its global interests -- to fly and fight in Air, Space, and Cyberspace" [30]. Shortly after adopting the new mission statement, the USAF announced plans for the creation of a new "Cyber Command" [31]. The Cyber Command will be comprised of personnel specializing in the intelligence, communications, operational, and engineering fields, with the stated goal to exploit cyberspace and to defend our national cyberspace.

1.1 Problem Statement

Delay-based IP Geolocation methods provide the ability to geolocate an IP addressable node based on network timing and routing information. Unfortunately, there has been no comparative analysis of the accuracy of existing delay-based IP Geolocation methodologies. For example, it is of interest to know how the positioning of nodes within the network would impact the accuracy of various IP Geolocation methods. Information gained from an analysis may enable an improvement of existing methods or lead to the development of a new or hybrid methodology.

1.2 Research Objectives

The purpose of the research is twofold. First, it is desired to gain a complete understanding of the accuracy of existing delay-based IP Geolocation methods. A simulation-based study of the delay-based IP Geolocation methods will allow validation of the existing IP Geolocation methods and provide a better understanding of the sensitivity the methods have to the network infrastructure. Second, it is expected that the

knowledge gained from the analysis will enable the development of a new hybrid methodology for delay-based IP Geolocation that may improve the accuracy when compared to existing individual methods.

1.3 Research Methodology

This research will make use of the OPNET network simulation tool which allows the user to rapidly create and simulate network architectures [14]. OPNET has been successfully used in numerous network studies and is considered one of the best tools for accurate simulations of real-world network architectures [21][14]. In this research, various network architectures will be modeled and simulated in OPNET to provide insight into accuracy of delay-based IP Geolocation methods.

The process for conducting the research will consist of four steps. First, a Wide Area Network architecture consisting of 12 nodes and 14 routers across the USA will be modeled in OPNET to enable the simulation of network traffic. Second, multiple simulations will be conducted to collect the propagation delay from each node to every other node. Third, an analysis of the collected data using the Upper-bound Multilateration (UBM), Constraint Based Geolocation (CBG), and Time to Location Heuristic (TTLH) methodologies will be conducted. The results will be analyzed to identify trends in the data and to investigate the effect of positioning nodes. Finally, a new methodology of combining multiple IP Geolocation techniques will be developed to obtain more accurate results than using one methodology alone.

1.4 Research Significance

Determining the geographic location of a network node has proven to be important in a wide variety of commercial applications. The full potential of the military application of this technology has not yet been realized, but is expected to grow with the new emphasis on cyberspace. For this reason, we need to understand the strengths and weaknesses of the delay-based IP Geolocation methods. Developing an enhanced understanding of the technology will help us protect the knowledge of the location of our network addressable assets and assist the military community in locating the positions of nodes of concern.

1.5 Thesis Overview

Chapter one has described the background of IP Geolocation and enumerated some of its potential uses. It introduced the importance of IP Geolocation and focused on some of the purposes and uses in commercial and military applications. This chapter also briefly introduced the problem statement, research objectives, research methodology, and explained the significance of the research. Chapter two contains a more in-depth review of literature related to the topic of IP Geolocation. It also explains the terminology, the underlying calculations required for the delay-based IP Geolocation methods, the metrics for comparing the IP Geolocation methods, and how the metrics are can be compared to evaluate the accuracy of the various methods. Chapter three provides an overview of the research design methodology used in this study. It explains the rationale behind the research design and its appropriateness to answer the research objectives. Chapter four presents an analysis of the data collected from the research, identifies trends discovered

in the data, proposes a new hybrid IP Geolocation methodology, and provides a comparative analysis of the three delay based geolocation methods. Chapter four also presents results from a proposed hybrid geolocation methodology. Finally, chapter five provides a discussion on the research results and presents implications for future research.

2. Literature Review

2.1 Chapter Overview

The purpose of this chapter is to expand upon the information presented in Chapter 1 through a detailed review of the relevant literature. First, a high level review of IP Geolocation will be addressed to provide a basic understanding of the various methods that have been used to determine the geographic location of an IP addressable node when using all available information. Next, a detailed discussion of the delay-based methods for IP Geolocation will be presented to provide an in-depth understanding of the operation, strengths, weaknesses, and limitations of each of the methods. Finally, a discussion of recent thesis research in the area of delay-based IP Geolocation will be presented to motivate the scope of this thesis.

2.2 Definition of Terms

In this section, terminology is introduced to provide the reader with the knowledge necessary to understand the research presented in this thesis. Initial descriptions of the terms will be basic and may be later refined as necessary to further clarify their application in the research.

A “PING” command is used on a IP addressable node to determine the delay for information to travel through the network from one point to another [19]. PING commands are often used to insure that another system on the network is reachable and is currently “up” or able to answer requests for service [11].

The terms “end node”, “target node” and “polling node” are used to refer to the role that an IP addressable node plays in the research. An end node is defined as a network device that will respond to a PING request and which the geographic location of the device is known. End nodes are primarily used when discussing the TTLH IP Geolocation method because it determines which end node is located closest to a IP addressable node with unknown location. For the purposes of this research, any IP addressable node that responds to a PING request can be an end node.

A target node is defined as any IP addressable node network device that will respond to a PING request and which the geographic location of the device is unknown. The whole purpose of IP Geolocation is to identify the geographic location of an IP addressable node to some level of granularity. Target nodes are used in the research to identify the node that the IP Geolocation is attempting to locate. For the purpose of this research, the location of target nodes will be known and used to calculate the accuracy of the various delay-based IP Geolocation methods, but in a real life application of the methods the location of the target node would be unknown.

A polling node is defined as any IP addressable node network device that can generate PING requests and receive the resulting response. The polling nodes are controlled by the researcher and are used to send PING requests to both end nodes and a target node of interest. Polling nodes are the primary data collection mechanisms in all delay-based IP Geolocation since these methods require delay measurements to determine the approximate location of the target node.

In this research project, all nodes will be used as polling nodes, end nodes, and target nodes. This allows for a research design in which all of the data can be collected and later analyzed by the researcher to create multiple scenarios for analysis.

2.3 Internet Protocol Geolocation

Internet Protocol Geolocation (IP Geolocation) is the process of identifying the approximate physical location of an IP addressable, networked device that is connected to an IP based network based upon its IP address and communication characteristics. In most cases, the IP based network contemplated for use is the Internet as this is the application environment of current interest. However, it can be used in any IP based network such as the Secure Internet Protocol Router (SIPR) or the Joint Warfare Internet Communications Secure (JWICS) networks.

IP Geolocation methods can be divided into three basic categories based upon the information that is used to determine the geographic location of the given target IP address. Database focused methods rely upon information contained in databases; information leakage focused methods rely upon information that is leaked interactions with the IP node of interest, and network communication attribute-focused methods rely upon network, routing, and timing information which can be collected and analyzed to identify the location of the target IP of interest to some level of granularity [12].

2.4 Database Focused Methods

IP Geolocation methods in the first category are based upon information that is stored in public database (e.g., a WHOIS database or a Domain Name Server database).

These methods use the target IP address in conjunction with information that is stored within a database containing information related to the network structure, architecture, or topology.

2.4.1 WHOIS

WHOIS databases contain information that maps logical identifiers (domain names, IP addresses, AS numbers contained in routing tables, etc.) to real-world entities (company names, ISPs, telecommunication providers, etc.), which could reveal the approximate geographic location of the target IP address. The WHOIS command queries a remote database that contains information recorded when a network domain or IP address range was registered. It is possible to cache WHOIS lookups in a local database to accelerate, optimize, or correlate searches. NetGeo is a for-profit application that relies primarily on WHOIS for its basis of information. Using information from 2,380 IP addresses, NetGeo was found to have a median error distance of 650 km [15]. While this is a very large area, it meets the needs for the majority of the commercial applications that NetGeo supports. A critical weakness of WHOIS-based methods is that the accuracy of this method is dependent upon the accuracy of the information stored in the database. The data contained within the WHOIS database could be erroneous because it was submitted incorrectly, changes have occurred since the information was entered, or by intentional deception. As a result, the accuracy of this information may not be reliable and should not be solely relied upon. However, WHOIS lookups can be used to compliment other IP Geolocation methods when there is a requirement for increase accuracy and reliability.

2.4.2 Domain Name Service (DNS)

The Domain Name Service (DNS) is a standard distributed database used to manage the forward and reverse translation of Uniform Resource Locators (URLs) to IP addresses. For example, when the URL “http://www.cnn.com” is entered into a web browser, the web browser first sends a request to the local DNS server asking for the IP address that corresponds to “www.cnn.com”. Once the local DNS server returns the IP address that corresponds to the URL, the computer can now communicate over the network using IP protocol to retrieve the desired web page [18]. When domains are created, information is provided by the registrant including the administrative contact and address, the technical contact and address, and the IP address of DNS servers that will be responsible for conducing DNS translations for the new domain. The addresses provided by the registrant can be a useful clue when conducting IP Geolocation. In some cases, the target node may be located in close proximity to the local DNS server. By identifying the DNS servers that are used by the target node, one can infer its location. Unfortunately, the geographic region served by the DNS server could be quite larger resulting in a coarse granularity in the identification of the area in which the target node is contained. Once again, this information should be used with information gained from using other methods.

2.4.3 IP Clustering

IP Clustering makes use of the principal of locality to conclude that IP addresses that are located in close proximity to each other may be physically located near to one another. If we know the geographic location of an IP addressable node (or groups of nodes) that is close to the IP address of a target node, we can infer the target node

location. In some cases, organizations build databases from previous queries to the WHOIS database and augment it from knowledge gained from other sources. This method relies on keeping a large database and that can be cumbersome because IP addresses are not always static. The internet is designed and organized in a way that allows for IP addressable nodes to dynamically change which makes it extremely difficult to insure that information previously collected is still valid [15]. In addition, this method will not be accurate in instances of organizations that interconnect their geographically dispersed locations using dedicated Wide Area Network (WAN) or Virtual Private Network (VPN) communication links. In these cases, a WHOIS query or other information may identify the target IP address as physically located at the organizations main headquarters while in reality it is located at a satellite office located very far away from headquarters. This occurs because the organization assigns IP addresses for their remote offices from the same block as those in the headquarters. This is a common situation in organizations with high level security requirements that mandate that the internal organizations be connected to the Internet via a single point. This security architecture is used to enable security monitoring, policy compliance, and auditing of all organizational traffic.

2.5 Information Leakage-Focused Methods

IP Geolocation methods in the second category are based upon information that is leaked from the IP address of interest during network communication interactions between network nodes. The information gained can be through legitimate mechanisms

designed into the communication infrastructure or can be inadvertently and unknowingly leaked from a target.

2.5.1 Reverse DNS

Reverse DNS lookups are used to convert an IP address into a system name and domain name. A DNS server database often has information about the geographic location of systems that they serve embedded into the system names. A reverse DNS lookup of an IP address can reveal the system name and/or domain name which may contain clues about its geographic location. These clues include the city, state, and region names; airport codes; or organizational names which can be used to identify location [12]. Administrators commonly use location in the naming of devices for ease of recognition of the location of the device. This is often done for efficiency so that when they encounter a problem with the node, they can find the troublesome device quickly [4]. DNS servers also optionally contain a record that identifies the latitude and longitude of the DNS server. The embedded information can provide valuable information aiding in IP Geolocation. However, this method is subject to deception and may not be reliable as discussed above due to the inherent dependence upon how the information is entered and maintained in the database.

2.5.2 Trace Route

A “TRACEROUTE” command is used to determine the current pathway by which information travels through the network from one point to another. The details of how TRACEROUTE works will be discussed in a later section. A TRACEROUTE command will identify all of the intermediate network routers that network communications pass through from the source (polling node) to the destination (target

node). Since the routers are often named based upon their physical location, their names can yield valuable clues about their geographic location. If one can locate the last router in the chain, one can infer that the target node is geographically near the last router.

Deception and social engineering can be used to entice a user to leak information. For example, consider a stealth counter intelligence web server which appears as a non-military, non-governmental resource that serves satellite imagery that is desirable to our adversaries. The user could be asked for their ZIP code or other unique geographic identifying information under the guise of enhancing the user's experience. Once this information is collected, it can be archived in a database for future use. The database can then be accessed as a compliment to other information resources when conducting IP Geolocation [29].

2.5.3 Application Information Leakage

Application programs can also leak information without the user's knowledge. For example, by design a web browser can provide information such as language, operating system, browser name, and time zone [18]. Organizations often leak clues about their location by their domain names. Some well-known examples are country code top-level domains such as .au (Australia), .de (Germany) or .uk (United Kingdom) [4]. While the granularity of these methods is coarse, it does provide useful and collaborating information. Other domains such as .mil, .edu, and .gov do not use a country code but they are all assigned within the United States. Interestingly, airport codes are often used in the naming conventions of network routers. By looking at the router names to which a user is connecting, clues to the location can be found [12].

2.5.3.1 Finding End-User IP Addresses

Java can be used to find the IP address of a computer that accesses a web page even when the user is connecting to the internet through a proxy [17]. A web page can use simple java applet programs that request a computer assessing the web page to make another connection to the web server [5]. That java enable connection requests information such as the IP address of the computer making the connection [12].

2.6 Network Communication Attribute-Focused Methods

IP Geolocation methods in the third category are based upon information collected using network, routing, and timing information. An analysis of IP Geolocation methods in this category are the sole focus of this thesis. Each of these methods make use of the delay for information to be sent and received between nodes. For this reason, the following discussion will discuss each of the factors which determine the delay through a network connection.

2.6.1 Delay Factors in IP Network Communications

Consider an ideal situation where network traffic passes from a sending node to a receiving node via a fixed path through the communication infrastructure elements (e.g., hubs, switches, routers, gateways, firewalls, communication links). Even in this simple case, the delay between two nodes is highly variable. The factors that contribute to the variation in delay include, but are not limited to, the network topology, switching speed, line speed, network loading, and queuing delays [3] [26]. Since all delay-based IP Geolocation methods use delay measurements as a means to determine geographic distance, it is important to understand these primary sources for delay.

2.6.1.2 Topology

Network topology is the physical infrastructure that the network traffic traverses and is typically thought of as the primary determinate of the overall delay. The network topology constrains the physical path by which the message can travel. The network topology can be categorized based upon the location of the given communication path element of interest in the chain from the source node to destination node [25].

Source and end node systems are usually connected to a Local Area Network (LAN) which is then connected to the Internet through the network infrastructure. LAN elements consist of hubs, switches, access points, and firewalls. Typically, a LAN is then connected to a Backbone Network (BN) within the organization that links together LANs. The BN is comprised of a collection of higher speed communication lines, servers, routers, and gateways [23]. In smaller organizations, the BN will connect to the Internet via an Internet Service Providers, (ISPs) Point of Presences (POP). The POP is the pathway by which organizational network traffic connects to the Internet. In larger organizations, the BN network may be connected to a larger network as discussed below.

A Metropolitan Area Network (MAN) is a network that covers the area of approximately the area of a city. MANs are used to interconnect BNs and to provide a connection to higher speed WANs. As a result, the equipment used in a MAN is often fiber based and transports network traffic at higher data rates [9]. MANs can be implemented using circuit switching, packet switching, frame relay, or asynchronous transfer mode communication links [23].

A Wide Area Network (WAN) is a network that spans a geographic area larger than that of a MAN. WANs are used to connect networks within a nation or

internationally. In most cases, computer network WAN traffic travels over the same point-to-point high speed communications links used for carrying digitized voice communications for telephone networks. WANs are typically implemented by using high speed point-to-point circuit switching, packet switching, frame relay, or asynchronous transfer mode communication links [23].

LANs, BNs, MANs, and WANs are interconnected to provide a physical path from the source node to the destination node. The network topology contributes to the delay because it defines the physical path through the network infrastructure that the message takes from source to destination. The complete path that the message travels is known as the "network distance". The network distance is determined by how the networks are interconnected within the organizations containing the nodes as well as how the organizations are connected to the Internet POP. For example, individuals in the Dayton, Ohio metropolitan area who use a Time Warner cable modem to connect to the Internet are connected first to the Time Warner internal network. The Time Warner internal network uses a combination of LAN, MAN, and WAN networking technologies to interconnect Time Warner customers across the state of Ohio. Time Warner has two POPs in Ohio: Columbus and Cincinnati. If a Time Warner cable modem user in Dayton connects to a web site located outside of the Time Warner private network, the traffic must first travel to either Columbus or Cincinnati where it enters the Internet POP. The Internet POP then routes the message to the appropriate location based upon its IP address. The route that the message takes can be over any of the numerous Internet providers that are connected to the POP.

2.6.1.3 Line Speed

Line Speed is a generic term used to describe the rate that data can travel through a given network segment via the specified medium. The medium acts as a conduit through which the message will travel. Depending on the location in the network hierarchy, different line speeds will be present. In the LAN, there is typically 10BaseT (10 Mbps) or 100BaseT (100 Mbps) hardwired networks. BNs employ higher speed systems with hardwired or optical networks running at 1000 Mbps or higher. MANs often employ high-speed digital fiber optic networks running from 1 Gbps to 40 Gbps. WANs use a variety of technologies including high-speed digital fiber optic networks running at speeds greater than 10 Gbps, microwave links can run at data rates of 274 Mbps, and satellite links that vary from 12 Mbps to 135 Mbps [23].

2.6.1.4 Queuing Delay

Network infrastructure elements such as routers often must handle a large amount of traffic during normal operation. When the router receives a message before it has had a chance to send a previous one, it will stack the messages awaiting transmission in a queue [18]. The result of a message being temporarily stored in a queue is delay. Many factors impact queuing delay such as the network congestion, amount of demand, and the processing speed of the network device. Excessive queuing delay can cause delay-based IP Geolocation methods to arrive at incorrect results. In most cases, this can be accounted for by sending multiple messages between the source and destination node over a variety of time, day, and weeks and at varying times of day over many days and choosing the minimum delay that is experienced. This method relies upon previous research that showed that there is a high probability that some of those messages reaching

the destination will not experience excessive queuing delays along the path [3]. Previous work determined that using the minimum time yielded better results than using averages [3] [21]. Finding the minimum response time is accomplished by sending many requests over a short period. Some of those requests will make it to the destination without waiting for retransmission. This delay is taken as the best-case scenario processing time by the end nodes and all of the network infrastructure elements. While this is not insured, the fact remains that the delay between two nodes can never be less than the delay through a given network under ideal conditions. For this reason, the minimum delay times are used instead of the average delay times. By using knowledge about the network, the network delay sampling times can be strategically selected to occur when the network is idle or under minimum loading conditions.

2.6.1.5 Switching Speed

Network traffic must propagate through the network infrastructure devices that connect the source and destination node. Network infrastructure equipment such as repeaters, routers, switches, hubs, or bridges result in a delay that is different than the contribution from network topology, line speed, or queuing delay [18]. The delay is a function of the design of the infrastructure element and the domain translation(s) which must occur to implement the functionality of the device. For example, fiber optic transmission systems must convert electrical signals to optical signals when transmitting data and must convert optical signals to electrical signals when receiving data. Since the connection from source to destination will traverse multiple network infrastructure devices, each one contributes to the overall delay through the network. For example, in a router even if there is no other traffic to cause a queuing delay, there is a finite amount of

delay required to receive data, process it, and send it out to another network interface. While switching delay can be significant, the NSA found that a packet travels through a switch in two milliseconds 95% of the time [8].

2.6.1.6 IP Path Diversity

The design of the Internet Protocol is such that Internet traffic does not necessarily follow the shortest path from origin to its destination. Furthermore, traffic does not always take the same path from source to destination [25]. This is an intentional design choice made to increase the resilience of the network to infrastructure element failures or excessive delays in infrastructure elements. The distributed nature of network routing decisions and adaptive routing algorithms means that network traffic will be routed based upon dynamic network and operational factors. The result is that network traffic from the same network connection does not always follow the same path from one packet to another [18]. Consider a network infrastructure where traffic is delivered faster when it travels from source to destination through three routers connected via high-speed links versus traveling through two routers connected via a lower speed link. Thus, the best quality connection does not always mean that it will traverse the least number of hops to the destination. Now, consider when one of the high-speed links between the routers becomes overly saturated resulting in excessive delay. The router nearest the source may decide to route the traffic via the lower speed connection because it is currently faster than the saturated high-speed link. Due to the dynamic nature of routing, it is important to characterize the delay between end nodes a number of times, over a variety of times of days, days of the week, and weeks during a month.

2.6.2 Measuring Delay

Previously, the PING command was briefly introduced as a tool for making delay measurements. PING works by sending an Internet Control Message Protocol (ICMP) Echo Request packet from the source node making the request to the destination node [11]. The outbound request sent by the source node has a message field that includes a timestamp that is used, in part, to calculate a round trip time once the echo request is replied and received [26][11]. When the receiving node receives a Echo Request packet and if it is configured to respond, it will respond to the request by sending a ICMP Echo Reply packet back to the sender. When received by the originating system, it can calculate the difference in time from when the message was sent to when the reply was received. This calculated time is known as the Round Trip Time (RTT). The RTT can be divided by two to estimate the one-way delay time from source to the destination. In this research, the delay time between nodes is calculated using this method. The PING command has a margin of error of four milliseconds [26][19]. In most cases, this is sufficient to diagnose network connections or determine the amount of loading that the infrastructure is currently experiencing.

In some cases, tools other than PING tools can be used to collect delay measurements when a finer resolution is required or precise time measurements in each direction are required. For example, some satellites network configurations use the satellite for transmission in one direction while the data is transmitted over terrestrial lines in the other direction. Unless accurate measurements are collected in each direction, asymmetric communication channels can give misleading delay measurements when using PING. One way to measure delay more precisely is to use the computer's

clock for measurement. In this case, the outgoing messages not only include a timestamp, but also a stamp that counts the number of computer cycles the processor uses between the sent and received transmission. Current generation computer processors running at more than 1 GHz allow for measurements of time to be in the microsecond level. This resolution is necessary when correlating delay measurements to distance calculations, especially when every microsecond could equate to a message traveling approximately 200 kilometers [7].

2.6.2.1 Trace Route

As previously discussed, the TRACEROUTE command is another tool that is frequently used by network administrators. The TRACEROUTE command also makes use of specially configured ICMP Echo Request packets. One of the header elements of an IP datagram is the Time To Live (TTL) field. The TTL field is an 8 bit, unsigned quantity that can range from 0 to 255. The TTL field was designed into the IP protocol to insure that all network messages will be dropped after a specified amount of time. Normally, the TTL field is initialized by the sending system to the maximum value (255), and is decremented by one as it passes through each of the network routers. If a router decrements the TTL field to 0 on an incoming network packet, it will drop the packet. This will insure that traffic traveling through an endless loop will eventually be dropped preventing network saturation due to erroneously designed or implemented networks. When a router drops a packet, it will inform the sender that it had to do so via an ICMP status message. This is what makes the TRACEROUTE command possible.

For example, when a TRACEROUTE command is issued, the sending system addresses the packet and sets the TTL to 1. The packet travels to the first router which decrements the TTL field to 0. Since the TTL is now zero, the router drops the packet and sends a status message to the sending system. The status message contains the IP address of the intermediate router. The sending system records the returned router message, and then sends a new packet to the destination with the TTL set to 2. The process above is repeated and the second router status message is recorded. The process is repeated until a trace from the source to the destination is completed.

The TRACEROUTE command shows the IP address of each device between the original and the destination node. A common use of TRACEROUTE is for an administrator to find which device is not responding in the network. A graphical version of the TRACEROUTE, known as Visual Route, works the same way TRACEROUTE does [32]. Visual Route adds a graphical user interface and also uses NetGeo to help determine the location of the devices. NetGeo uses information entered by administrators to determine the locations of the routes taken [25].

2.7 Delay-Based IP Geolocation Methods

In this section, existing methods for delay-based IP Geolocation are reviewed. Delay-based IP Geolocation provides an attractive means for conducting geolocation because it does not rely upon the accuracy of information entered by others and delay measurements are not easily corrupted, neither purposely or inadvertently, as database methods can be. There is a significant gap in the academic literature when comparing the

accuracy of existing IP Geolocation methods. For this reason, this thesis focuses upon an examination of delay-based IP Geolocation methods.

2.7.1 Upper-Bound Multilateration

Initially, delay-based IP Geolocation methods were focused upon multiplying one-half the RTT by some scale factor to determine the distance between the polling node and the target node. By PINGing from a number of different geographically separated nodes, one can draw circles on a map around the polling nodes and see where the circles intersect which (ideally) indicates the geographic location of the target IP address. Upper-bound Multilateration [15] is a method that uses triangulation to find the location. While triangulation is the common term, multilateration is the more accurate name. Triangulation refers to an angle-based methodology when using three reference points [7]. Multilateration uses more than three reference points to further refine the estimation of the target location.

There is a common notion that the correlation between delay and actual distance is poor [2]. One reason for this is that network traffic does not travel in straight lines nor even in the most direct path to the host. However, some research has found success in countering this notion. To accomplish correlating distance to delay, first the one-way delay time must be determined. That time is then multiplied by 2/3 the speed of light to find an upper bound on distance that the target is away from the polling station. The rate of 2/3 the speed of light was the average speed that network traffic was found to travel through various media [17]. With the use of three polling stations, the researcher can look

at the distances and conclude that the target resides in the area where the distances overlap as shown in Figure 1 below.

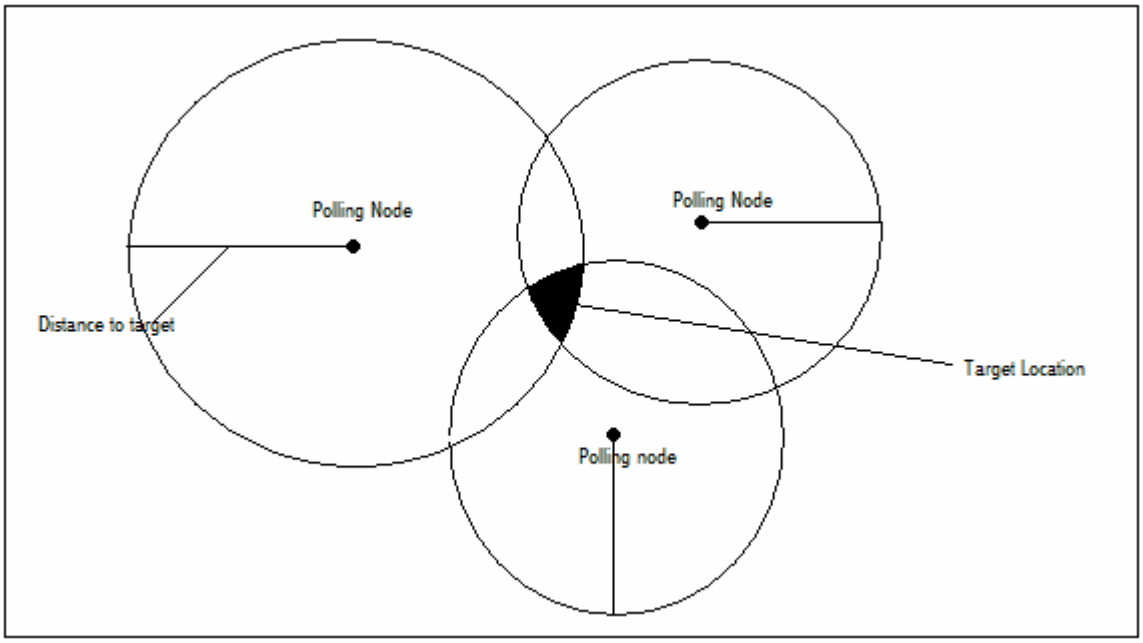

Figure 1. Delay Triangulation IP Geolocation Method

One interesting side effect of this method is that it has an error checking algorithm built into it. When the upper bound is underestimated, the intersecting area will not include the target location. If the correct location of the target was in the area of each circle, the circles will always have an area of overlap. By plotting the areas, a mistake can be found if the circles do not overlap at some point. This may, in fact, be the best feature of the method because it is an error checker for the data collection [12].

Research has shown that correlating actual distance to delay is reasonably accurate when the distances are large in scale such as from an area the size of a continent. Research has shown that the method has been able to correctly find the location of a target to within 100s of miles [7]. The messages that were sent and measured traveled long distances and the distance traveled was a large factor in the delay. In geolocation methodologies that research nodes from a smaller area, messages travel shorter distances therefore factors such as queuing delay, and switching speed play a bigger factor in the equation. The distance traveled is still a factor in the delay, but not as prominent as it is when traveling hundreds or thousands of miles.

2.7.2 Constraint-Based Geolocation

Constraint-Based Geolocation is a method that was proposed that recognized that the rate of transmission is not always a constant 2/3 speed of light [7]. This improved upon the previous method using a calculated scale factor which depended upon the network infrastructure between the polling node and other known end nodes. Each polling node would poll a number of geographically dispersed nodes at known distances and then calculate a scale factor based upon the delays. The delay to the target node would then be multiplied by the scale factor and the circle drawn around the polling node. This process is repeated for each of the polling nodes within the polling node set.

To accomplish IP Geolocation by this method, the location of a variety of end nodes must be known. Once that is determined, the location can be found by comparing the delay times of the target node with that of the known end nodes. The end node with the delay time nearest to the target will be the end node that is closest to the target.

Methods such as GeoTrack use this method [15] while other research projects use variations of this method [8][32]. While this method does not allow you to pinpoint exactly where the target is, it can give an approximate location which is good enough for the intended application.

Some other conclusions can be made by correlating delay to distance. The speed of light can be used as the absolute upper bound for determining distance. Traveling at the speed of light, it would take a message 134ms to circumvent the equator. Internet traffic travels slower than the speed of light so if the round trip time of a message is less than 67ms one can conclude that the target is located on the same side of the globe. Network traffic can also travel through satellites positioned in geosynchronous orbit. Those satellites are located 22,000 miles from earth in outer space. A message traveling at the speed of light would take 478ms to travel to a satellite and back to earth. Messages that have round trip times shorter than 478ms did not use a satellite to reach the destination [21].

Previous research has had success correlating an actual distance to delay on a large scale. The accuracy of the previous CBG research was able to find the location of the target to within 120 kilometers [7]. An accuracy rate of 120km is a good result when looking at areas starting at the size of a country.

2.7.3 Nearest Known Node

The National Security Agency (NSA) holds a patent on a methodology for locating devices by using delay measurements known as the Time To Location Heuristic (TTLH) method [8]. The methodology attempts to find the nearest end node with a

known location to the target node. The result is not a precise location but rather the location of the previously known node that is closest to the target. The nearest end node is determined by polling the delay from multiple locations and comparing the delay measurements from the known nodes with that of the target node. The comparison is done using the Euclidean distance formula. The primary motivation for development of this method was to account for delays introduced by line speed, switching speed and queuing delays. By identifying the nearest node, many of the variations in delay are normalized because nodes in close proximity to the target will experience similar delay phenomena.

TTLH works best over long distances when the delay is dominated by line speed delay. When switching delay is dominate, such as within a campus or across a city, TTLH is not as accurate because the delay spent in switches becomes a larger factor in the delay than the network topology. TTLH uses multiple polling nodes and requires multiple end nodes with known location that are in proximity to the target node. In fact, the granularity of the method is determined by the distance between the end nodes and the target IP address.

The TTLH method builds upon research that found a correlation between delay time and location [8]. The time to location approach finds the proximity of a location by using a set of polling nodes and end nodes, then comparing delay times to determining which end node is in closest proximity to the target. To establish what the delay is, PING messages are sent from the polling nodes to the end nodes and to the target node. The RTT is then charted.

In order to account for delay from line speed and switching speed a hypothetical 0-bit packet is sent [3] [26]. This is accomplished by sending multiple messages at sizes of 64, 128, 512, and 1024 bits. Next the latency is charted on a graph and the y intercept is used for a hypothetical 0 bit packet, as shown in Figure 2 [3] below.

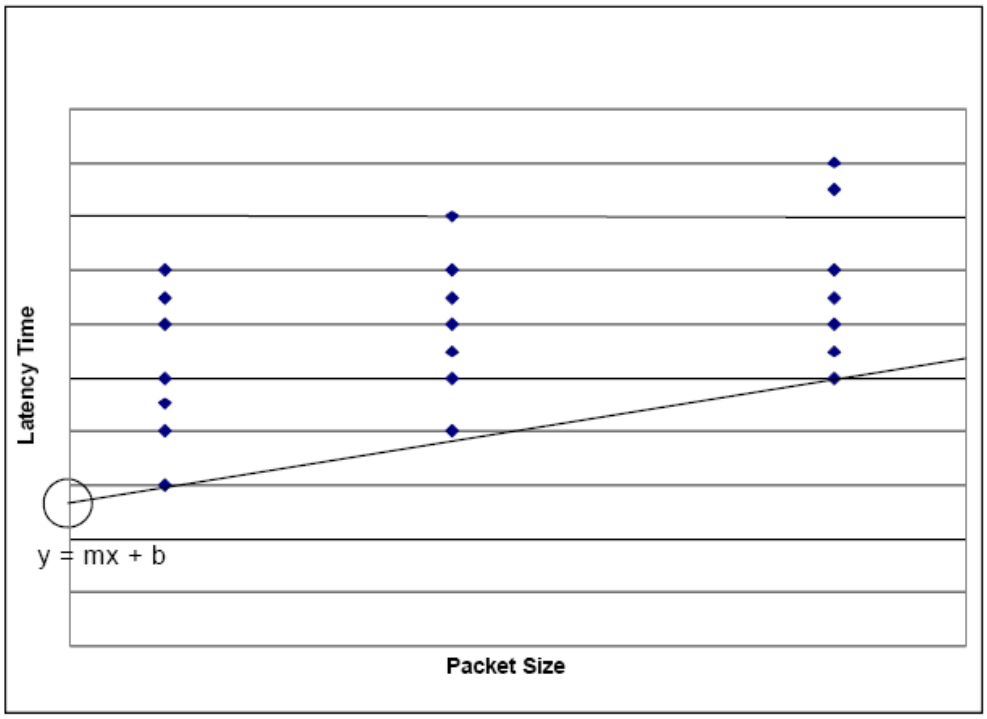

Figure 2. Hypothetical Zero-Bit packet [3]

The minimum time is used for each packet size to determine the best fit. The resulting theoretical "zero-byte" packet would take zero time to switch and zero time to transmit through a medium thus resulting in a measurement that takes out the delay factors of line speed and switching speed [26].

The quickest time is used in the 'zero-byte' calculation to account for queuing delay. The packets are sent multiple times over a short period of time and collected

during different times of the day and over many days. Theoretically, some of these messages will get through without being held in a queue. The messages that do not experience a queuing delay will be used for the calculations. Previous research has found that the messages sent at times when network loads are small have the smallest queuing delay and result in the best data [26]. A traffic analysis of the network of interest is required to select the ideal time to collect measurements. If this is not available, a uniform sampling across a 24 hour period can be conducted and the minimum time selected. Messages sent at other times of the day should have more network traffic to compete with, but if the background traffic is consistent, the TTLH method will still produce the correct result.

2.7.3.1 Euclidean Distance

The Euclidean distance formula shown in Equation 1, is used to find the nearest known node to the target. The equation uses a limit but instead of finding the greatest distance as the limit approaches one, it can be used to find the nearest location as the result is closer to zero.

$$d_e = \sqrt{\sum_{i=1}^{p} (x_{ie} - x_{it})^2} \qquad (1)$$

The Euclidian Distance formula is used to find the nearest known end to the target node. The Euclidean distance is the square root of the sum of the squares of the delay between the end node and target node. In this formula, *e* represents an end node. This

equation is solved for each end node. The end node, or *e*, with the lowest value is the closest end node in the dataset to the target. The equation is solved by using the delay from all of the polling nodes to one end node and comparing that with the time from the polling nodes to the target. X_{ie} is the time for the message to travel from the polling node to the end node, X_{it} is the time from the polling node to the target node, p is the number of polling nodes used in the TTLH example. This is repeated for every end node. Then it concludes that the end node with the smallest Euclidean distance is the end node that is closest to the target. The results of this equation result in a number, but that number cannot be directly converted to a distance. An example of this is shown below in Figure 3.

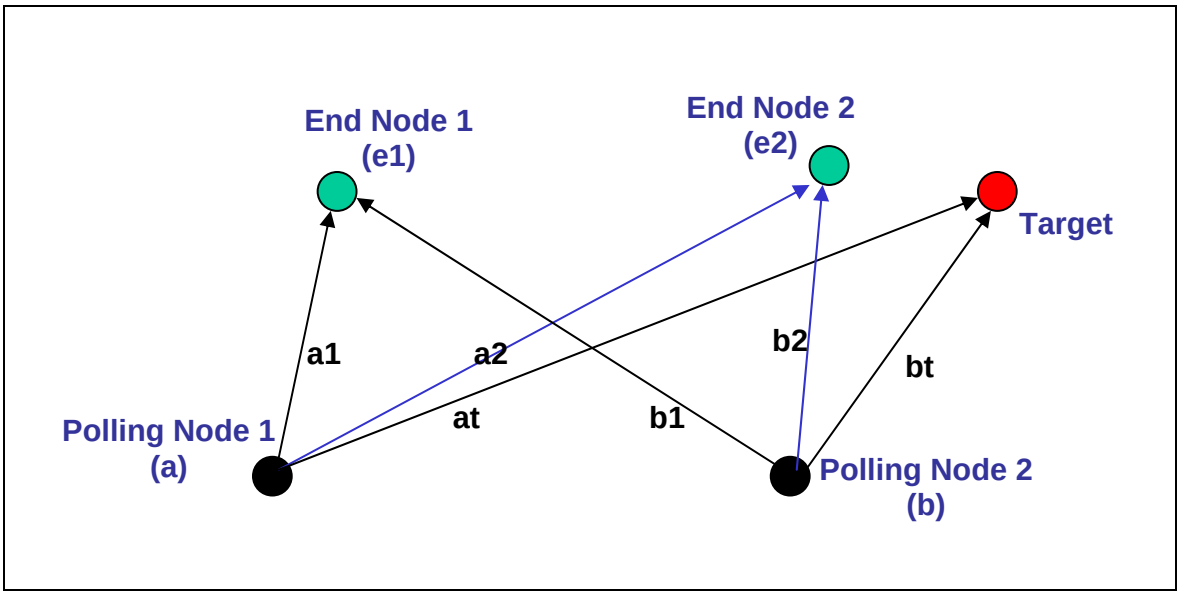

Figure 3. Euclidean Distance Methodology

Polling Node 1 sends requests to end node 1, end node 2, and the target. These delay times are represented by a1, a2, and at. Polling Node 2 also sends a request to end node 1, end node 2 and the target. Those delay times are represented by b1, b2, and bt. Next, by using this simple data set, Euclidean distances could be found as shown below in Equation 2:

$$d_1 = \sqrt{(a1 - at)^2 + (b1 - bt)^2}$$

$$d_2 = \sqrt{(a2 - at)^2 + (b2 - bt)^2} \quad (2)$$

Then once the equation was solved if d2 was less than d1, we could conclude that d2 (end node 2) is closer in location to the target than d1.

Simulation and experimentation have been used to measure the effectiveness of the TTLH method over a variety of network architectures and scales. Clarson [3] showed that TTLH worked within a building by correctly identifying to which switch in the building the target node was connected. He first simulated the local area network in OPNET. Then he conducted a controlled experiment was carried out on a LAN. The results of the experiment showed the location of the target was accurate only when an end node was located on the same switch as the target. This result was the first indication that TTLH was not reliable when the network distance was small.

Sorgaard [18] conducted simulations in OPNET to show how multiple autonomous system (AS) networks would affect the results when using the TTLH method. His simulations showed that the method was accurate in a single AS, but was

only successful 71.4% of the time in a multiple-AS network. In his work, models of the underlying infrastructure of the MCI and AT&T networks were constructed within OPNET to allow the simulation of the network environment. It was hypothesized that TTLH would have to be modified to account for link speed between AS in order for the results to be successful more often. Unfortunately, no mechanisms were proposed to account for link speed variances.

Turnbaugh conducted an experiment where he used publicly available "looking glass" servers that are located throughout the United States [26]. He was able to control them to poll target nodes and end nodes located throughout the country. The results correctly identified which end node city was closest to the target 100% of the time for five out of the six targets. The sixth target was a case that had multiple end nodes located within 100 miles of the target node. However, the TTLH method did not return accurate information when multiple end nodes were located near the target node. It correctly identified the nearest known node when the target node only had one end node located within 100 miles of it. When conducting the experiment, polling sets of seven, eight, or nine nodes were used because previous researchers theorized that these numbers of polling nodes would allow for the most accurate measurements [15].

2.8 Summary

This chapter provided a review of the relevant IP Geolocation literature. It first examined existing methods for determining the geographic location of IP addressable nodes. While IP Geolocation is accomplished through a variety of methods, delay-based methods use real-time delay measurements and are the least dependent upon external

information sources. A review of the terminology used in the thesis was presented to provide the reader a basis for interpreting the research design. In addition, a review of computer networking architectures and factors that contribute to network delay was required so that the reader would understand the research.

While the literature review revealed three methods for delay-based IP Geolocation, there is no existing research which compares the accuracy of each of the methods. For this reason, this thesis will focus on a comparative analysis of delay-based IP Geolocation methods.

3. Methodology

In this chapter, the complete research design is reviewed to provide the reader a detailed understanding of how the research data will be collected and analyzed. First, a brief overview of the OPNET network simulation package that will be used in the modeling and simulation of the networks is presented. Second, the network architecture that will be modeled in OPNET and used for data collection is introduced. Third, an overview of the data collection process is reviewed to provide the reader the ability to duplicate the results. Fourth, the calculations required for Upper-bound Multilateration (UBM), Constraint-Based Geolocation (CBG), and Time to Location Heuristic (TTLH) IP Geolocation methods investigated in this project are explained in detail. Finally, the metrics by which the accuracy of the methods will be compared is presented along with all assumption made in the analysis.

3.1 Simulation Tools

The OPNET network simulation package provides the ability to rapidly model and simulate network architectures [14]. OPNET has been successfully used in a variety of network studies and is considered one of the best tools for accurate simulations for real-world network architectures [21][14]. One key benefit of OPNET is that it offers a number of predefined templates that model a variety of different network architectures. All models in OPNET can easily be parameterized to provide the simulation of realistic, non-ideal networks across a variety of operational situations [14].

Each of the network architectures used in the research will be modeled and simulated using OPNET version 12.0 running on a Dell Precision Workstation 690N - 1KW. dual core Intel Xeon® Processor running at 3.0GHz with a 4MB L2 cache and 4 GB of system RAM.

3.2 Network Architecture

A Wide Area Network (WAN) architecture model based upon an OPNET template will be used as a basis for the simulation and data collection in this research. A WAN network was chosen because the literature review revealed that the delay-based methods for IP Geolocation worked well when the geographic distance between nodes was large. The network model consists of fourteen routers graphically dispersed across the United States of America which interconnect twelve polling nodes located in different cities throughout the United States as shown in Figure 4. The nodes contained in the model were located in Boston, Hartford, Houston, Jacksonville, Los Angeles, Miami, New Orleans, New York, San Diego, San Jose, Seattle, and Tampa. While the network architecture was fixed, each of the twelve nodes could be considered as an end node (E), polling node (P) or the target node (T) depending on the specific simulated model under consideration. A listing of the twelve cities and their node designators are shown in Table 1.

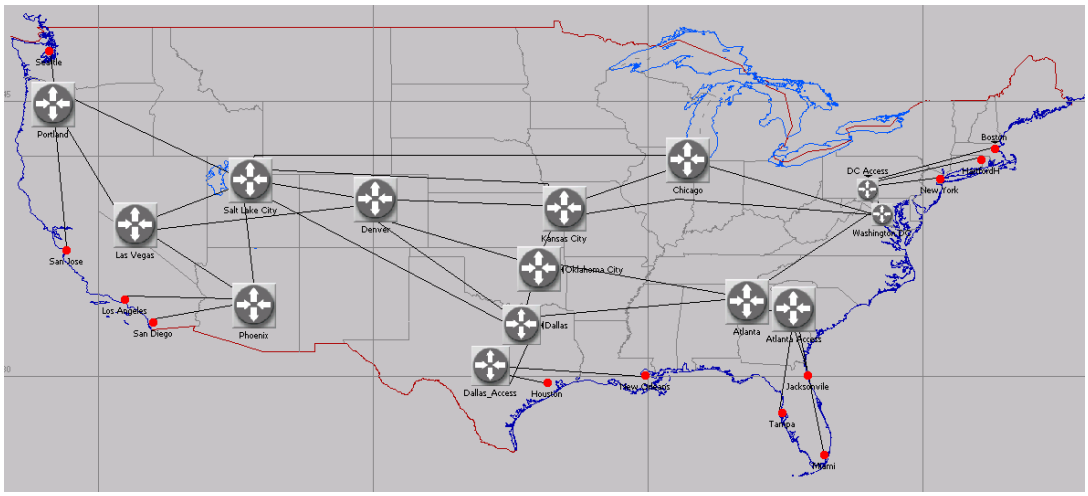

Figure 4. Network Topology

The initial research focus is upon the impact that network topology has when conducting delay-based IP Geolocation. As a result, the modeled network architecture is selected so that there are a variety of distances between nodes to enable the investigation of various network architectures without requiring the explicit reconfiguration of the model. Each network link simulates using an OC-3 link. The network is configured to be unloaded (e.g., no background traffic) to prevent queuing delays. Switching speed is assumed constant in this research because of the limited number of network nodes. Further, the literature review revealed that almost all messages travel through a switch in less than two milliseconds [8].

Once an analysis of impact network topology has upon the accuracy of the delay-based IP Geolocation methods, if time permits analysis of other factors will be conducted.

Table 1- Node Designators

City	End Node	Polling Node	Target Node
Boston	E1	P1	T1
Hartford	E2	P2	T2
Houston	E3	P3	T3
Jacksonville	E4	P4	T4
Los Angeles	E5	P5	T5
Miami	E6	P6	T6
New Orleans	E7	P7	T7
New York	E8	P8	T8
San Diego	E9	P9	T9
San Jose	E10	P10	T10
Seattle	E11	P11	T11
Tampa	E12	P12	T12

3.3 Enumerating Network Architectures

While the network architecture is constrained by only containing twelve nodes, there is a large number of possible subsets of the overall network architecture that will be used in the analysis phase of the research. The process by which each unique network configuration is as shown in the pseudo code shown below in Figure 5:

```
for(Target=1; Target <=12; Target++)
{
   for(NumPolling=2; NumPolling<=9; NumPolling++)
   {
      Enumerate All Combinations of NumPolling nodes
      {
         Enumerate All Combinations of (11-NumPolling) End nodes
         {
            /* analyze data */
         }
      }
   }
}
```

Figure 5. Network Architecture Enumeration

Once a network architecture is enumerated, the delay data will be analyzed using each of the three delay-based IP Geolocation methodologies.

3.4 Data Collection Process

Since each of the nodes can serve as a polling node, an end node, and a target node during any given simulation, delay measurements for all possible combinations of the twelve nodes will be collected at one time. Data will be collected by configuring the OPNET simulation so that every node PINGs each of the eleven remaining nodes. Since data will be collected for each node, a 12 by 12 matrix will be constructed which contains 132 points of data as shown below in Equation 3:

$$Delay = \begin{bmatrix} 0 & D_{1,2} & D_{1,3} & . & . & . & . & . & . & . & D_{1,11} & D_{1,12} \\ D_{2,1} & 0 & & & & & & & & & & D_{2,12} \\ D_{3,1} & & 0 & & & & & & & & & . \\ . & & & 0 & & & & & & & & . \\ . & & & & 0 & & & & & & & . \\ . & & & & & 0 & & & & & & . \\ . & & & & & & 0 & & & & & . \\ . & & & & & & & 0 & & & & . \\ . & & & & & & & & 0 & & & . \\ . & & & & & & & & & 0 & & . \\ D_{11,1} & & & & & & & & & & 0 & D_{11,12} \\ D_{12,1} & . & . & . & . & . & . & . & . & . & D_{12,11} & 0 \end{bmatrix} \quad (3)$$

Note that the delays on the diagonal are zero due to the delay between any node and itself is zero. For completeness, the delay between any two nodes is not assumed to be symmetric. This is an important reality that must be accounted for in the analysis

phase to account for modern asymmetric network architectures. Asymmetry can come from differences in network architecture on the return path (e.g., satellite downlink with terrestrial return path) or differences in line speed. One limitation of using PING is that the delay measured is the Round Trip Time (RTT) which must be divided by two to estimate the one-way distance between the polling node and the target node. In an asymmetric network, this estimation may induce errors into the findings.

Once the delay data is collected, it can then be analyzed in multiple ways by theoretically changing the numbers and locations of each of the polling, end and target nodes. Specifically, the data will be analyzed for each possible unique network configuration as identified in the enumeration of network architectures section above.

3.5 Data Analysis Process

In this section, the process by which each of the delay-based IP Geolocation methods will be applied to the collected delay data is reviewed in detail. Specifically, the calculations required for Upper-Bound Multilateration (UBM), Constraint-Based Geolocation (CBG), and Time to Location Heuristic (TTLH) methods is reviewed. A computer program was written in the "C" programming language to make all of the calculations necessary for each of the delay-based IP Geolocation methods (see Appendix A).

3.5.1 Upper-Bound Multilateration (UBM)

The UBM method requires that the delay between two nodes be multiplied by a constant, 2/3 the speed of light (~124,000 mi/sec), to estimate the distance between the

polling node and the target node. The round-trip time collected in the data collection phase will be divided by two to estimate the one-way delay between the polling node and the target node. The resulting product results in a pessimistic of the distance between the polling node and target node [17]. Equation 4 shows how the calculation to estimate the distance will be calculated:

$$\text{EstimatedDistance}_{\text{UBM}} = \left(\frac{2}{3}c\right)\left(\frac{1}{2}RTT\right) \tag{4}$$

Once the estimated distance using the UBM method is calculated, the estimated distance to other nodes to the target node can be calculated and circles drawn around each of the polling nodes. The intersection of the circles provides the multilateration necessary to localize the geographic location of the target node. For example, if there is a delay of 0.03083 between two nodes A and B, using the UBM we would obtain the result shown in Equation 5 below

$$\text{EstimatedDistance}_{UBM} = (124188.16)(0.015415) = 1914.36 \text{ miles} \tag{5}$$

3.5.2 Constraint-Based Geolocation (CBG)

The CBG method is similar to the UBM method in that it multiplies a rate times a delay. However, instead of using a fixed constant rate, it requires the calculation of the Best Line rate based upon analysis of the delay data involving all nodes other than the target node. The calculated rate Best Line Rate is then used to estimate the distance between the polling node and the target node. Once again, the round-trip time collected

in the data collection phase will be divided by two to estimate the one-way delay between the polling node and the target node. The resulting product results in a distance that is less pessimistic estimate of the distance between the polling node and target node when compared to the UBM method [7]. Equation 6 shows how the calculation to estimate the distance will be calculated:

$$\text{EstimatedDistance}_{\text{CBG}} = \left(BestLineRate\right)\left(\frac{1}{2}RTT\right) \tag{6}$$

The Best Line must first be calculated for each polling node when considering each of the other nodes as a target and the remaining nodes as end nodes. For this reason, there are theoretically 132 unique best lines, and hence 132 unique Best Line rates. However, in reality many of the Best Line rates will be the same because the Best Line is determined either by two nodes or by a single end node and the origin. The Best Line is determined by finding a line that is close to, but below all data points and has a non-negative intercept. If a proposed Best Line would cross the y-intercept at a negative number, the origin point (0, 0) was used to avoid any negative distance measurements. The slope of the line will determine the Best Line rate used to estimate the distance based upon the delay. Once the line is drawn and the x intercept is determined, the Best Line rate can be calculated by using any point on the line as shown in Equation 7 below.

$$\text{BestLineRate} = \left(\frac{\left(y-b\right)}{x}\right) \tag{7}$$

For example, consider how the Best Line rate is calculated for Boston (P1) as the polling node and Tampa as the target node (T12). In this case, the delay between the polling node (e.g., Boston) and all other end nodes (e.g., Hartford, Houston, Jacksonville, Los Angeles, Miami, New Orleans, New York, San Diego, San Jose, Seattle) is plotted with distance on the x-axis and delay on the y-axis as shown in Figure 6. The Best Line is the line which fits below all of the plotted points, but does not have a negative y-intercept. In this case, the y-intercept is positive (approximately 0.0075) and thus the given line is the Best Line.

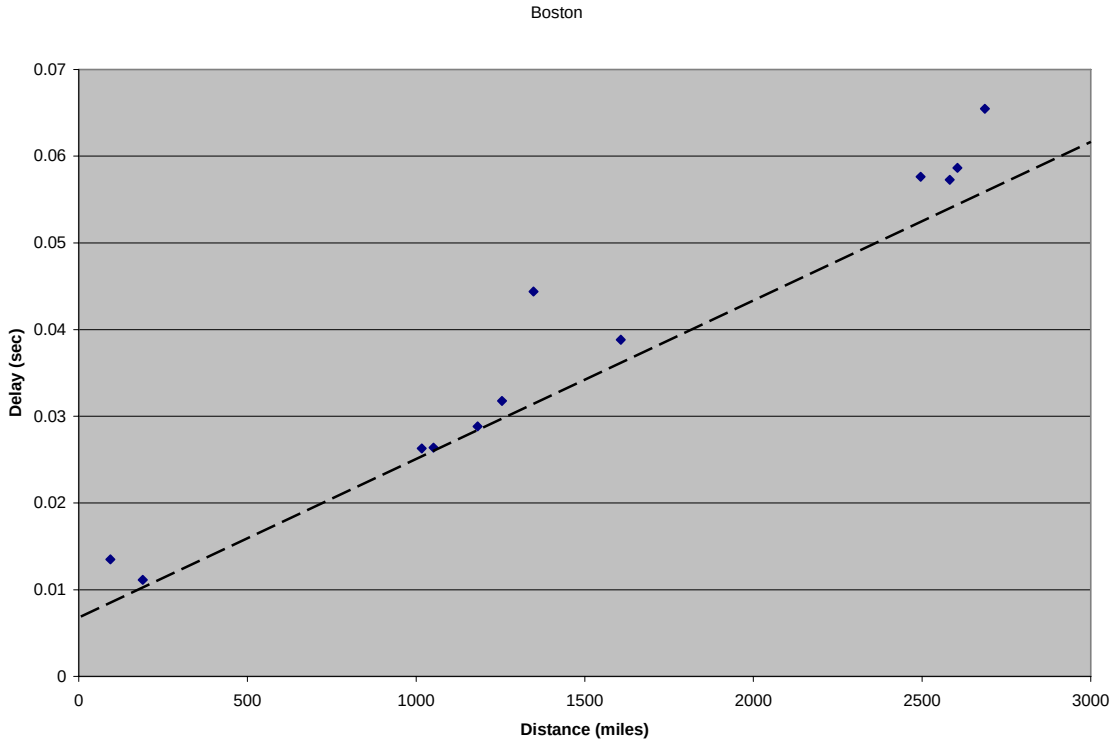

Figure 6 - Determination of the Best Line Rate for Polling Node Boston (P1) and Target Node Tampa (T12)

In this example, the slope of the line is calculated as shown in Equation 8 below by using any point on the Best Line (e.g., 500 miles, 0.015 seconds):

$$\text{BestLineRate}_{P1,T12} = \left(\frac{\left(0.015 - 0.0075\right)}{500} \right) = 0.000015 \text{ seconds/mile} \quad (8)$$

Note that in some cases, one or two data points used to plot the Best Line actually lies on the Best Line. In these cases, the Best Line rate used is the pessimistic 2/3*c* as defined for the UBM method. This will account for biasing the results of the analysis by erroneously indicating the exact correct location when geolocating a node whose data was used to define the Best Line.

In some cases, the Best Line would result in a negative intercept, which is not allowed. In these cases, the origin point (0,0) was used as an anchor point in conjunction with one other data point to define the Best Line. Figure 7 shows an example using Seattle as the polling node and Boston as the target node. In this case, it was necessary to use the origin (0,0) and the San Jose node to determine the Best Line.

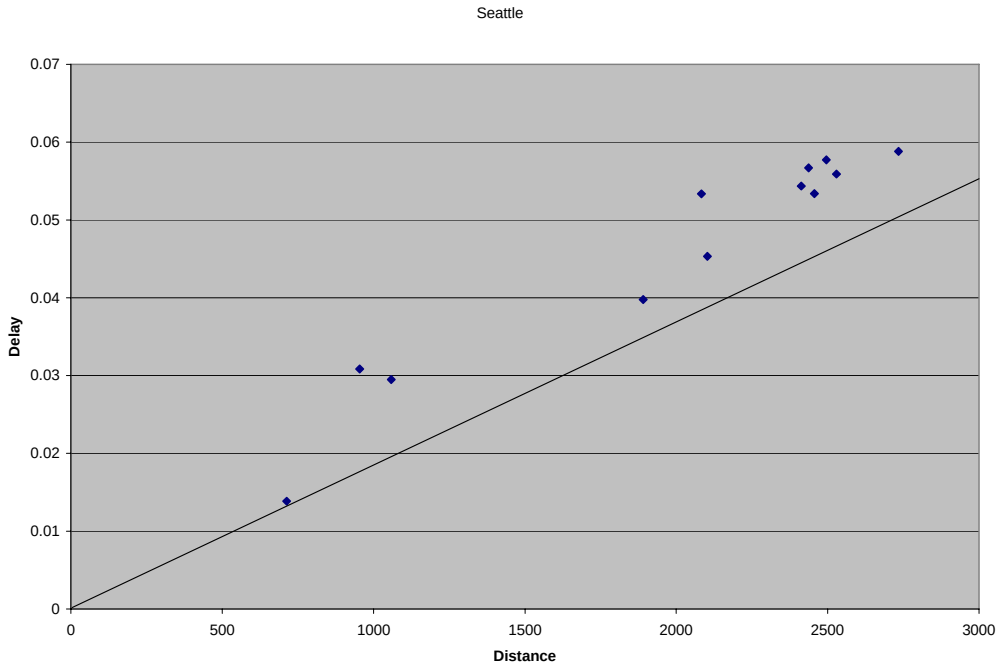

Figure 7. Seattle (P11) Bestline

Once the Best Line rate is calculated, the estimated distance from all polling nodes to the target node, circles are drawn around each of the polling nodes. The intersection of the circles provides the multilateration necessary to localize the geographic location of the target node as was shown in the UBM method.

3.3.3 Time to Location Heuristic (TTLH)

The TTLH method is unique in that it does not resolve to a distance from the polling node, but instead identifies the node nearest to the target. The TTLH method uses the Euclidean distance as a means to locate the end node that is nearest to an unknown target. The round-trip time collected in the data collection phase will be divided by two

to estimate the one-way delay between the polling nodes and the end nodes, as well as between the polling nodes and the target node. These values are entered into the Euclidian Distance formula shown below in Equation 9 where p = number of polling nodes, e = end nodes, t = target node, x_{ie} = time from polling node to end node, and x_{it} = time from polling node to target node:

$$d_e = \sqrt{\sum_{i=1}^{p} (x_{ie} - x_{it})^2} \tag{9}$$

Suppose that there are 3 polling nodes, 4 end nodes, and 1 target node. The calculations resulting from the application of the Euclidean Distance formula would be as follows as shown in Equation 10 below:

$$\begin{aligned} d_1 &= \sqrt{(x_{11} - x_{1t})^2 + (x_{21} - x_{2t})^2 + (x_{31} - x_{3t})^2} \\ d_2 &= \sqrt{(x_{12} - x_{1t})^2 + (x_{22} - x_{2t})^2 + (x_{32} - x_{3t})^2} \\ d_3 &= \sqrt{(x_{13} - x_{1t})^2 + (x_{23} - x_{2t})^2 + (x_{33} - x_{3t})^2} \\ d_4 &= \sqrt{(x_{14} - x_{1t})^2 + (x_{24} - x_{2t})^2 + (x_{34} - x_{3t})^2} \end{aligned} \tag{10}$$

In the TTLH method, the end node (d_e) with the lowest magnitude would be identified as the node nearest to the target.

Since TTLH requires at least two polling nodes to solve the Euclidean Distance equation, the number of possible end node combinations is limited. There are eight ways to organize the polling, end, and target nodes from the set of nine possible end or polling nodes as shown below in Table 2. During each iteration of the TTLH, the maximum number of end nodes will be included in the test. This is done to include as many data

points as possible when calculating the results. Two polling nodes, nine end nodes and one target can be permuted in any one of 660 different ways. The eight different possible categories of network architecture and the corresponding number of unique network architectures possible are shown below in Table 2:

Table 2 - TTLH Research Design

Number of Polling Nodes	Number of End Nodes	Number of Target nodes	Number of Possible ways to calculate
2	9	1	660
3	8	1	1,980
4	7	1	3,960
5	6	1	5,544
6	5	1	5,544
7	4	1	3,960
8	3	1	1,980
9	2	1	660

3.4 Metrics

In this section, the figures of merit used for measuring the accuracy of each of the delay-based IP Geolocation method is presented.

3.4.1 UBM and CBG

Metrics will be gathered for the purpose of comparing the accuracy of UBM to CBG. Since the UBM and CBG methods estimate a distance between the polling node

and the target node, important metrics include the miss distance, the percent of error, and the target overlap area.

The first metric that will be calculated is the miss distance. The miss distance is defined as the difference between the estimated distance to the target and the actual distance to the target as shown in Figure 8 below. For example, suppose that Miami is the polling node and the CBG method estimated the distance to the Hartford target to be 1,914 miles. However, in reality the target is 1,414 miles away. In this case, the resulting miss distance would be calculated as (1,914-1414) = 500 miles.

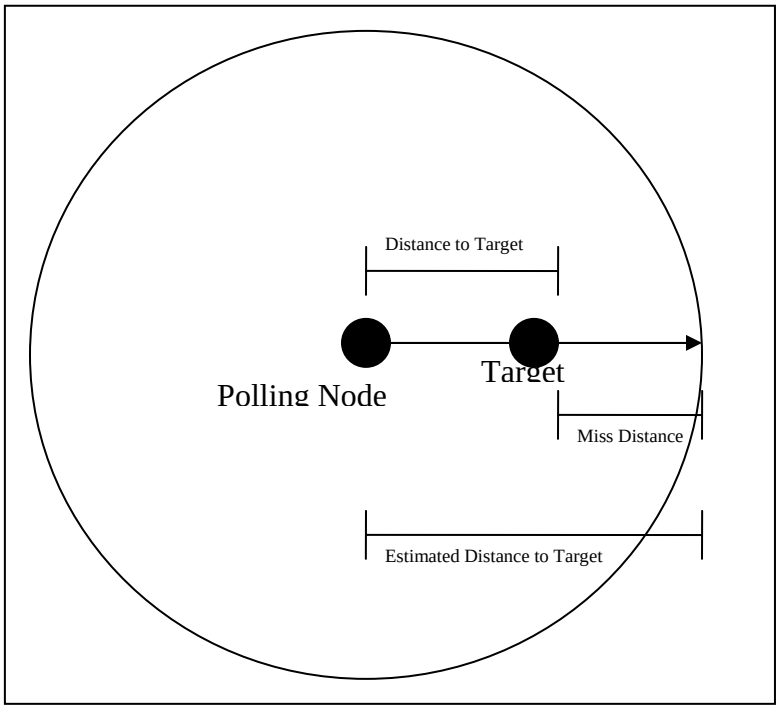

Figure 8. Miss Distance

Note that a positive result for miss distance will show that the target location is being underestimated. Both of the UBM and CBG methods are designed to be pessimistic and should never underestimate the distance from the polling node to the target node.

This is important because the area of a circle drawn around a polling node that underestimated the distance to the target would not cover the target. The miss distance will be calculated for each of the twelve polling nodes to all of the eleven targets, for a total of 132 calculations.

The second metric that will be calculate is the percentage of error. The percentage of error is defined as the ratio of the miss distance to the actual distance to the target time one-hundred as shown in Equation 11:

$$PercentofError\left(\%\right)=\left(\frac{\text{Miss Distance}}{\text{Actual Distance}}\right)*100 \tag{11}$$

Using the example from Miami to Hartford again, the miss distance to the target was 500 miles from the polling node but the actual distance is 1,414 miles. Equation 12 shows the result of the calculating the percent of error from Miami to Hartford is 0.3536.

$$PercentofError_{Miami-Hartford}=\left(\frac{500}{1414}\right)*100=35.36\% \tag{12}$$

Note that if the miss distance is small, the percentage of error drops. The percentage of error will be calculated for the UBM and CBG methods for each of the 132 cases discussed above considering each node as a polling node and the remaining nodes as target nodes.

The third metric that will be calculated is the area of overlap. The area of overlap is calculated by integrating the area that is overlapped by the intersecting circles surrounding the polling nodes as shown in Equation 13 below:

$$Area=\int_{MinLat}^{MaxLat}\int_{MinLong}^{MaxLong} f_{UBM}(x,y)dydx \tag{13}$$

Since the area of overlap is not easily represented as a continuous function due to its irregular shape, a computer program was written in the "C" programming language to calculate the discrete approximation to the area (see Appendix B). A two dimensional grid is created and laid on top of the geographic map. The grid has 1 mile resolution in both the x and y dimensions. Step functions are created that serve to locate the circles surrounding each polling node and return True if a given point is within the circle, and False otherwise. The step functions are logically ANDed together to determine if for a given x and y coordinate, the circles overlap. The area is of overlap is calculated by sweeping across the x and y dimensions and summing the number of points that overlap. Equation 14 below shows the equivalent area calculation:

$$Area = \sum_{MinLat}^{MaxLat} \sum_{MinLong}^{MaxLong} f_{CIRCLE1}(x,y) \cap f_{CIRCLE2}(x,y) \cap \ldots f_{CIRCLEN}(x,y) \qquad (14)$$

The overlap area was found 12 times, once for each of the targets when all of the polling nodes were utilized. Figure 9 shows an example of what the area of overlap might look like for a target with five polling nodes.

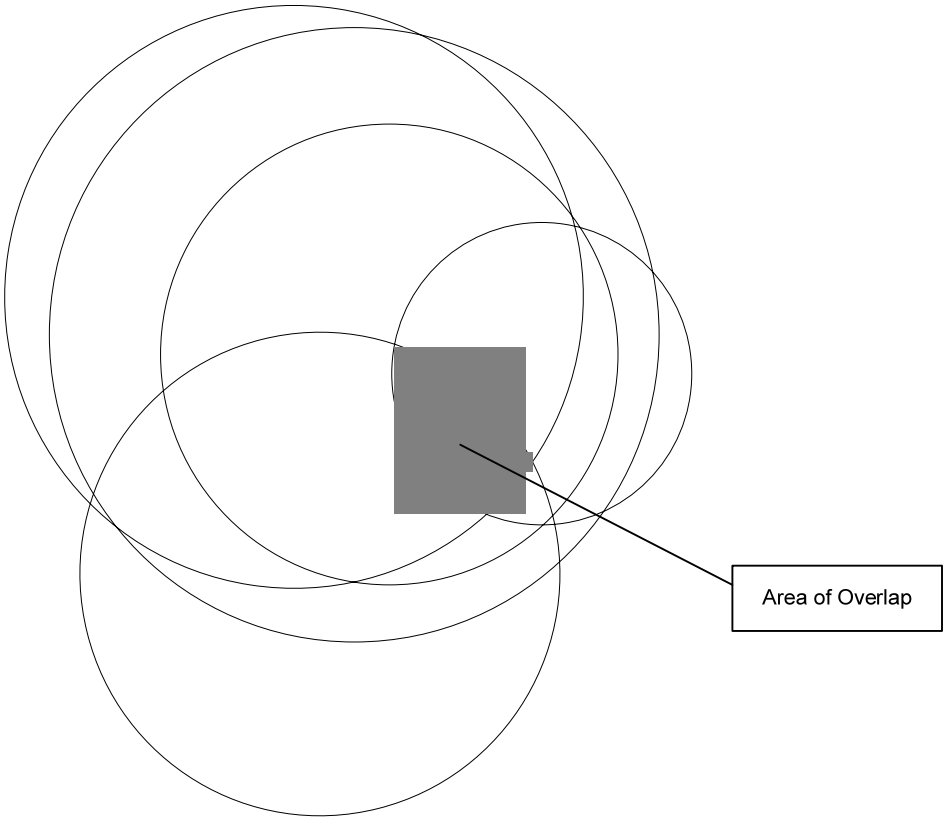

Figure 9. Area of Overlap

3.4.2 TTLH Metrics

The TTLH strives to find the nearest end node to the targets location. The UBM and CBG methods return the targets area, but the TTLH returns a node. Comparing the resulting area from UBM and CBG to the resulting node in TTLH is like comparing apples to oranges. The goals of the TTLH are different from the goals of UBM and CBG. Therefore, the metrics collected from the TTLH results do not attempt to directly compare the accuracy against the accuracy of UBM and CBG, rather the TTLH metrics will show the likelihood of the nearest known-node being selected.

The results of TTLH will be considered a success if the nearest-known end node is correctly identified. It is a failure if any node other than the nearest node is identified as the nearest node. The accuracy of the TTLH is limited by the number and the location of end nodes. The results have a finite number of possible results for the nearest known node. The TTLH examples were considered to be a success when the result showed the nearest-known network node to be the same nearest-known node physically. A computer program was used that could calculate the Euclidean distances for all combinations of the TTLH. The program stored and counted the number of successful and failed trials for the TTLH calculations.

The Euclidean distances for every possible combination of each iteration were computed with the use of a computer program. The program stored the results of each iteration. It did this by producing files for the successful and the unsuccessful trials. The Number of trials in those files were counted to find the accuracy rates. Those successes and failures were analyzed to see if the numbers and locations of the end and polling nodes have an effect on the accuracy of the results. The TTLH methodology was tested by calculating the results a total of 24,288 times, one for every possible iteration of the project as shown in Table 2.

3.5 Assumptions

A real-world application of the delay-based IP Geolocation methods discussed in this thesis would require some assumptions to be reported along with any analysis using these methods. For example, it is assumed that the target node has not tried to elude location detection. An evasive target employing any number of methods could invalidate

the results of the analysis by corrupting the delay measurement. For example, the target computer can add a fixed delay or vary the fixed delay using a distribution and parameters designed to confuse the analysis. This technique could be built into any piece of network equipment such as a router or a switch.

3.6 Summary

This chapter introduced the research design, the data collection process, and how data will be analyzed to answer the research questions. A discussion of how the network architectures will be permuted from the basic network map was presented. Examples of the application of the UBM, CBG, and TTLH geolocation methods was provided to give the reader insight into the analysis required to answer the research questions. Metrics were defined in an effort to measure the accuracy of the delay-based IP Geolocation methods.

4. Results and Analysis

In this chapter, the research data collection, application of the delay-based IP Geolocation methods, metrics calculation, and an analysis of the results are presented. Based upon the findings of the research, a novel hybrid methodology is proposed that can be used to improve the accuracy of delay-based IP Geolocation methods.

4.1 Data Collection

The network architecture presented in chapter three was modeled using OPNET version 12.0. With OPNET, a command script was written which allow for automated data collection. The script resulted in each of the 12 nodes in the architecture to PING each of the remaining 11 nodes every 30 minutes. The results from the PINGs over a 24 hour time period were written into a file. While a single PING would be sufficient, the researcher wanted to verify that the delay times did not vary when using ideal network configurations. The model was simulated on Dell Precision Workstation 690N - 1KW Dual Core Intel Xeon® Processor running at 3.0GHz with a 4MB L2 cache and 4 GB of system RAM. Upon completion of the simulation, the collected delay measurements were compared and the ideal simulation showed less than 0.01% difference in delay measurements across the 24 hour simulated time interval.

The collected delay measurements were entered into an Excel spreadsheet to facilitate post-processing and save in a comma delimited text file. Subsequently, the delay measurement file was loaded into a “C” computer program where it was stored as a 12 x 12 matrix of floating point number for use when calculating the results using each of

the geolocation methodologies under test. The delay data collected for this research project is shown in Table 3.

Table 3 - Delay Time (seconds)

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
E1	0.00000	0.01340	0.03878	0.02617	0.05881	0.03186	0.04446	0.01102	0.05736	0.06560	0.05772	0.02881
E2	0.01350	0.00000	0.03788	0.02540	0.05785	0.03083	0.04357	0.01013	0.05629	0.06467	0.05669	0.02795
E3	0.03882	0.03800	0.00000	0.02758	0.04101	0.03303	0.01286	0.03561	0.03951	0.04770	0.03976	0.03006
E4	0.02629	0.02528	0.02754	0.00000	0.05443	0.01497	0.03311	0.02299	0.05315	0.06112	0.05338	0.01180
E5	0.05865	0.05785	0.04114	0.05447	0.00000	0.05992	0.04676	0.05536	0.01346	0.03870	0.03084	0.05695
E6	0.03178	0.03077	0.03309	0.01483	0.05994	0.00000	0.03868	0.02858	0.05862	0.06671	0.05882	0.01733
E7	0.04438	0.04369	0.01300	0.03315	0.04670	0.03863	0.00000	0.04122	0.04508	0.05331	0.04533	0.03562
E8	0.01114	0.01013	0.03553	0.02287	0.05550	0.02848	0.04122	0.00000	0.05396	0.06228	0.05436	0.02551
E9	0.05726	0.05635	0.03963	0.05297	0.01332	0.05856	0.04522	0.05396	0.00000	0.03724	0.02950	0.05546
E10	0.06546	0.06467	0.04784	0.06118	0.03870	0.06663	0.05347	0.06216	0.03728	0.00000	0.01386	0.06366
E11	0.05762	0.05675	0.03992	0.05326	0.03077	0.05882	0.04548	0.05423	0.02945	0.01378	0.00000	0.05575
E12	0.02883	0.02776	0.03008	0.01178	0.05693	0.01947	0.03561	0.02553	0.05570	0.06366	0.05590	0.00000

4.1 Upper-Bound Multilateration

The first method analyzed based upon the simulation results was Upper-Bound Multilateration (UBM). UBM finds an upper bound of the distance to a destination node by using 2/3 speed of light for a rate of transmission. The Round-trip Time (RTT) from each polling node was first divided in half to estimate the one-way transmission delay, that delay measurement was then multiplied by 2/3 speed of light to find the upper bound for the target node. In this research project, each polling node used the eleven other nodes as targets. The resulting distances from each node to all of the other nodes are shown in Table 4.

Note that the driving distances were used as the actual distances between targets. As discussed in section 2.6, network traffic flows along fiber-optic lines that are primarily buried along right-of-ways which typically parallel major highways. For this reason, the driving distances between nodes are shown Table 5 were used as the physical distance between nodes.

The first metric used to measure the accuracy of the UBM method, the miss distance, is used as a sanity check because it determines if the use of the 2/3 rate was too slow. The miss distance metric was calculated by subtracting the actual distance between cites from the distance that UBM had estimated it to be. If UBM underestimated the upper bound, the result of subtracting the actual distance from the estimated distance would be a negative number. The results were positive for all 132 iterations of the project. The number of miles that were overestimated to each node are shown in Table 6.

Table 4 - Estimated Miles to Targets UBM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
E1	0.00	831.75	2407.70	1625.25	3651.45	1978.44	2760.77	684.53	3561.97	4073.37	3584.20	1788.99
E2	837.96	0.00	2352.37	1577.44	3592.39	1914.49	2705.44	629.26	3495.53	4015.63	3520.24	1735.72
E3	2410.68	2359.83	0.00	1712.49	2546.48	2050.78	798.34	2211.36	2453.28	2961.89	2468.99	1866.24
E4	1632.52	1569.99	1709.82	0.00	3379.72	929.55	2055.63	1427.73	3299.99	3795.19	3314.46	732.65
E5	3641.51	3592.39	2554.80	3382.21	0.00	3720.49	2903.27	3437.47	835.60	2402.86	1914.92	3536.01
E6	1973.48	1910.76	2054.51	920.86	3721.74	0.00	2401.68	1774.71	3639.77	4142.18	3652.38	1075.90
E7	2755.80	2712.89	807.04	2058.11	2899.55	2398.88	0.00	2559.46	2798.89	3309.99	2814.60	2211.86
E8	691.98	629.26	2206.39	1420.28	3446.41	1768.50	2559.46	0.00	3350.79	3867.35	3375.50	1583.96
E9	3555.76	3499.25	2460.73	3289.06	826.91	3636.05	2807.59	3350.54	0.00	2312.14	1831.84	3443.43
E10	4064.68	4015.63	2970.58	3798.92	2402.86	4137.21	3319.92	3859.71	2314.62	0.00	860.44	3952.66
E11	3577.55	3523.53	2478.49	3306.82	1910.76	3652.56	2824.10	3367.61	1828.73	855.47	0.00	3461.56
E12	1790.24	1723.73	1867.48	731.41	3534.77	1208.79	2210.99	1585.20	3458.46	3952.66	3470.94	0.00

Table 5 - Driving Distance between Nodes

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
E1	0	101	1854	1560	2986	1511	1530	215	3044	3135	3083	1360
E2	101	0	1754	1062	2906	1414	1429	117	2930	3043	2991	1262
E3	1854	1754	0	871	1550	1187	349	1631	1471	1889	2443	982
E4	1560	1062	871	0	2420	351	546	946	2342	2758	3023	204
E5	2986	2906	1550	2420	0	2736	1897	2781	121	341	1137	2531
E6	1511	1414	1187	351	2736	0	863	1297	2658	3076	3363	280
E7	1530	1429	349	546	1897	863	0	1306	1819	2236	2723	657
E8	215	117	1631	946	2781	1297	1306	0	2803	2942	2890	1146
E9	3044	2930	1471	2342	121	2658	1819	2803	0	461	1258	2452
E10	3135	3043	1889	2758	341	3076	2236	2942	461	0	839	2869
E11	3083	2991	2443	3023	1137	3363	2723	2890	1258	839	0	3133
E12	1360	1262	982	204	2531	280	657	1146	2452	2869	3133	0

Table 6 - Number of Miles Overestimated by UBM

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
E1	0	730.7509	553.6999	65.25176	665.4453	467.4432	1230.767	469.5257	517.9677	938.3749	501.1974	428.994
E2	736.9603	0	598.374	515.4393	686.3938	500.4862	1276.441	512.2619	565.527	972.6274	529.2404	473.7172
E3	556.6804	605.8253	0	841.494	996.4803	863.7828	449.3442	580.3583	982.277	1072.89	25.9868	884.2391
E4	72.51677	507.988	838.824	0	959.7214	578.5491	1509.626	481.7303	957.9925	1037.193	291.4605	528.6486
E5	655.5103	686.3938	1004.801	962.2052	0	984.494	1006.273	656.4689	714.6007	2061.857	777.9209	1005.012
E6	462.4756	496.7606	867.5085	569.8559	985.7359	0	1538.677	477.7123	981.7716	1066.175	289.3767	795.905
E7	1225.8	1283.893	458.0374	1512.11	1002.547	1535.883	0	1253.458	979.8929	1073.99	91.60272	1554.855
E8	476.977	512.2619	575.3907	474.279	665.4105	471.5029	1253.458	0	547.7855	925.3466	485.499	437.9592
E9	511.7583	569.2526	989.7283	947.064	705.9075	978.046	988.5861	547.5372	0	1851.137	573.8389	991.43
E10	929.6817	972.6274	1081.583	1040.919	2061.857	1061.208	1083.925	917.709	1853.621	0	21.43836	1083.664
E11	494.5533	532.5314	35.4872	283.8229	773.7606	289.563	101.1031	477.613	570.7342	16.47083	0	328.5615
E12	430.2359	461.733	885.481	527.4068	1003.77	928.7864	1553.986	439.201	1006.457	1083.664	337.9377	0

4.1.1 UBM Results

There were three metrics gathered for UBM to quantify the accuracy of the UBM estimated distance from a polling node to a target node: the miss distance, the error rate, and the area of the overlapping circles.

Miss distance was calculated by subtracting the actual distance from the UBM estimated distance. All of the miss distance figures were positive, which indicates that UBM never underestimated the distance. The mean miss distance for all 132 iterations was 770.69 miles. The maximum miss distance was 2,061.86 miles and the minimum miss distance was 16.75 miles when using the UBM method. The miss distance for all 132 iterations of the UBM is shown in Table 6.

The percentage of error was calculated by taking the miss distance and dividing by the actual distance. The mean error percentage for all of the 132 iterations was 99.6%. On average, this methodology is missing the target by a factor of 2. However, the variance was large with some estimations having error rate of 1% while others were greater than 700%. The error rates for all 132 iterations of the UBM is shown in Table 8.

The area of overlap was calculated twelve times, once for each target. The area was calculated by integrating the overlapped circles using all other eleven nodes as polling nodes. The results for all of the targets showed a mean area of 1,315,534 square miles. To put the size of that area into perspective, the approximate size of the state of Texas is 261,797 square miles [28]. The smallest area of overlap was 117,315 square miles, and the largest area found by UBM was 2,465,820 square miles. Table 7 shows the area of overlap for each of the twelve targets. Figure 10 shows the resulting area for

the target of Hartford (T2) when all of the other nodes were utilized. Note that some of the upper-bounds from a couple of end nodes are so large that they do not appear on the map.

Table 7 - UBM Results (Area)

Target Node	Area (mi^2)
Boston (T1)	1,014,887
Hartford (T2)	798,751
Houston (T3).	1,047,164
Jacksonville (T4)	1,214,409
Los Angeles (T5)	1,72,8845
Miami (T6)	2,465,820
New Orleans (T7)	2,012,245
New York (T8)	467,384
San Diego (T9)	1,458,443
San Jose (T10)	1,937,909
Seattle (T11)	117,315
Tampa (T12)	1,523,236

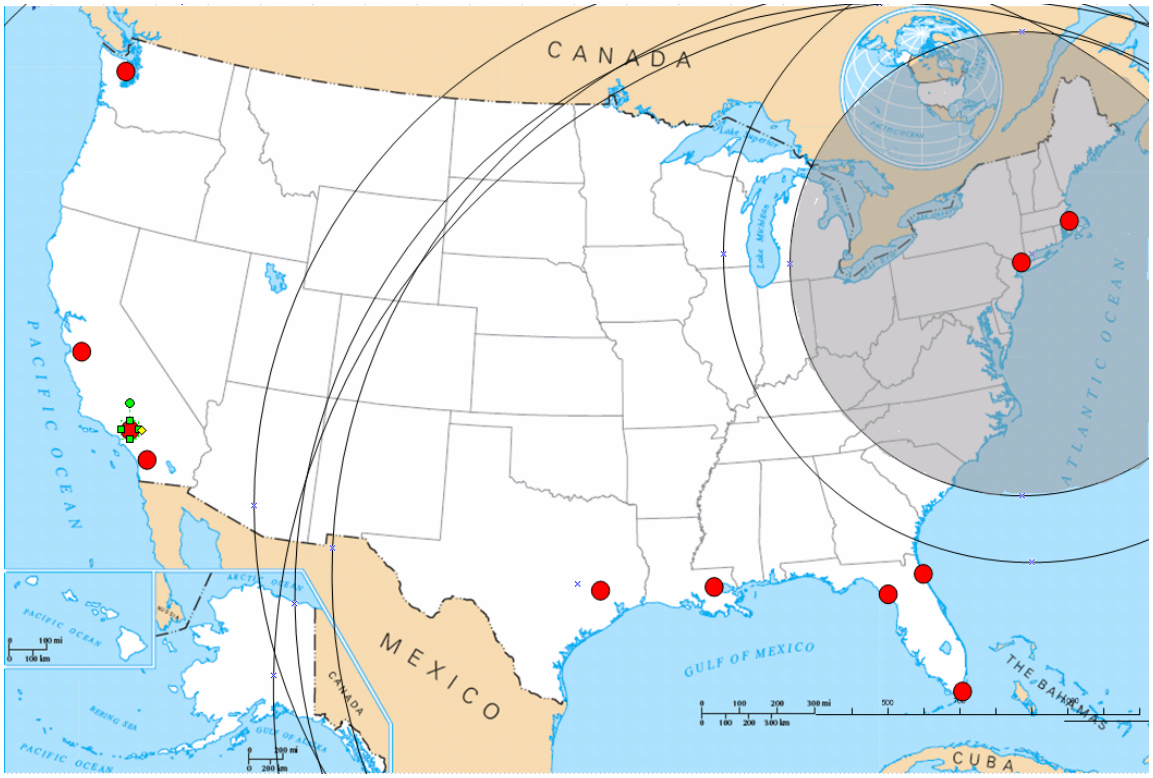

Figure 10. UBM results for Hartford (T2) target.

Table 8 - UBM Error Rates

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
E1	0	7.235157	0.298652	0.041828	0.222855	0.30936	0.804423	2.18384	0.17016	0.299322	0.162568	0.315437
E2	7.296636	0	0.341148	0.485348	0.236199	0.353951	0.893241	4.378307	0.193013	0.319628	0.176944	0.37537
E3	0.300259	0.345396	0	0.966124	0.64289	0.727702	1.287519	0.35583	0.667761	0.567967	0.010637	0.900447
E4	0.046485	0.478331	0.963059	0	0.396579	1.648288	2.764883	0.509229	0.409049	0.376067	0.096414	2.591415
E5	0.219528	0.236199	0.648259	0.397605	0	0.35983	0.530455	0.236055	5.905791	6.0465	0.684187	0.397081
E6	0.306073	0.351316	0.730841	1.623521	0.360284	0	1.782939	0.368321	0.369365	0.346611	0.086047	2.842518
E7	0.801176	0.898455	1.312428	2.769432	0.528491	1.779702	0	0.959769	0.538699	0.480317	0.03364	2.366598
E8	2.218498	4.378307	0.352784	0.501352	0.23927	0.363533	0.959769	0	0.195428	0.31453	0.167993	0.382163
E9	0.16812	0.194284	0.672827	0.404383	5.833947	0.367963	0.543478	0.19534	0	4.015482	0.456152	0.404335
E10	0.296549	0.319628	0.572569	0.377418	6.0465	0.344996	0.484761	0.311934	4.020869	0	0.025552	0.377715
E11	0.160413	0.178045	0.014526	0.093888	0.680528	0.086103	0.037129	0.165264	0.453684	0.019632	0	0.104871
E12	0.31635	0.365874	0.901712	2.585327	0.39659	3.317094	2.365275	0.383247	0.410464	0.377715	0.107864	0

4.2 Constraint-Based Geolocation

An analysis of the Constraint Based Geolocation results showed that it was better than UBM for every metric collected. This was an expected result because CBG uses attempts to use a more accurate rate of transmission than the UBM method. The rate of transmission is found by first determining a bestline from each polling node. Table 11 shows the equation of the bestline ($y = mx + b$) that was used from each polling node to each target node.

4.2.1 CBG Results

Miss distance was calculated by subtracting the actual distance from the CBG estimated distance. All of the miss distance figures were positive, just like the UBM results, proving that CBG never underestimated the distance from the polling node to target. That means that the target was always located in the overlapping area of the circles. The mean miss distance for all 132 iterations was 661.11 miles, slightly better than the UBM results. The maximum miss distance was 3,459 miles and the minimum miss distance was 10 miles when using the CBG method. The miss distance from each polling node to all of the targets is shown in Table 9.

To find the percentage of error, the estimated distance was divided by the actual distance. The mean error of the 132 iterations was 78.4%. The maximum percentage of error was 1014.37% and the minimum percentage of error was 1.43% when using the CBG method. Again, the maximum error rate was from the San Jose polling node to the Los Angeles target node. The error rates all 132 iterations are shown in Table 10.

The third metric collected from the resulting UBM data was the area of overlap. As in UBM, the area was calculated twelve times, once for each target. The results for all of the targets showed a mean area of 687,913 square miles. The most accurate target resulted in an area of 25,785 square miles, the largest area found by CBG was 1,668,759 square miles. The individual areas for each target are shown in Table 12. Figure 12 shows the resulting area for the target of Hartford (T2) when all of the nodes were utilized. CBG proved to be more accurate than UBM. The effectiveness of each method was calculated three separate ways. CBG was the more accurate method in all three cases.

The maximum miss distance in the CBG method was from polling node in San Jose to the target node of Los Angeles. As seen in figure 4, the network path that a message would have to take in this simulation would have required the message to travel from San Jose through routers in Portland, Las Vegas, then Phoenix before it could reach the target nodes location in Los Angeles. The network topology forced an indirect route to the target.. The network topology also had an effect on the rate used for the bestline. As discussed in section 3.5.2, The San Jose polling node required the use of the data point (0,0) when drawing the bestline in order to keep al results positive. Figure 11 shows the bestline for the polling node in San Jose. The one data point used to find the bestline is located in Seattle. This was because the network topology allowed for a direct route through one router to the node in Seattle, but it did not allow for any direct routes to any other nodes.

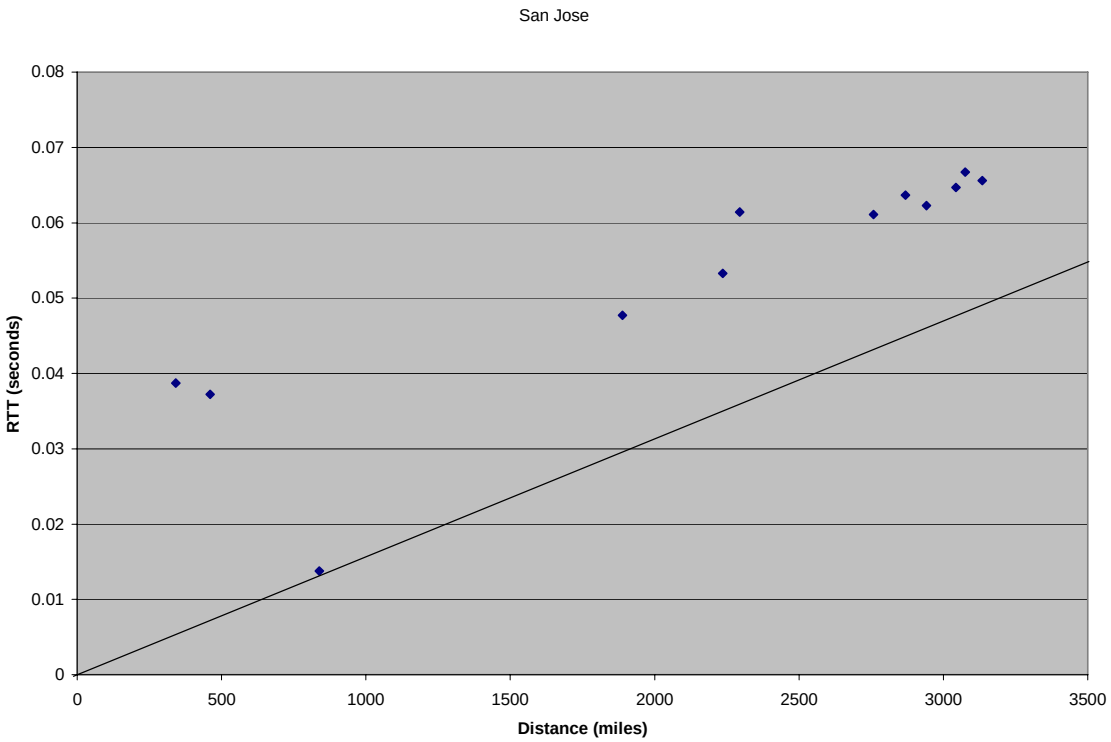

Figure 11. San Jose (P10) Bestline

Figure 11 demonstrates an unexpected advantage of the CBG method. By charting the delay vs. the distance for all of the end nodes, the outlier for one case can be seen. Even with the outlier included the upper-bounds are not underestimated. The CBG method defaults to a pessimistic upper-bound and the target are is included in the area. CBG defaults to include the target in a large area of overlap instead of underestimating the target and not including it in the area. The success of the CBG in this project could be slightly improved by excluding the outlier in the overall calculations. In order to accurately compare the CBG method with the UBM method all 132 calculations were included. Even with an outlier clearly visible and included, CBG always resulted in more accurate data.

Table 9 - Number of Miles Overestimated by CBG

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
E1	0	349	571	65	39	289	1170	10	518	965	567	240
E2	349	0	571	488	69	336	1171	83	95	957	634	288
E3	896	171	0	804	375	713	449	169	454	1161	82	668
E4	73	113	654	0	330	474	1304	54	458	1142	377	396
E5	1264	219	1050	1030	0	689	953	294	104	3459	838	719
E6	589	86	788	499	364	0	1387	53	542	1124	387	645
E7	1620	821	458	1529	353	1387	0	869	481	1189	1152	1343
E8	477	512	519	454	665	328	1144	0	548	1008	585	254
E9	1156	95	1004	1008	706	717	906	1057	0	1939	1617	698
E10	1365	507	1211	992	1409	749	1564	558	1339	0	21	781
E11	1217	59	35	377	88	290	101	110	117	16	0	329
E12	490	462	718	421	394	820	1368	439	598	1081	417	0

Table 10 - CBG Error Rate

Error Rate = CBG miss distance /// distance												
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**
E1	0	3.455446	0.307983	0.041667	0.013061	0.191264	0.764706	0.046512	0.170171	0.307815	0.183912	0.176471
E2	3.455446	0	0.325542	0.45951	0.023744	0.237624	0.819454	0.709402	0.032423	0.314492	0.211969	0.228209
E3	0.483279	0.097491	0	0.923077	0.241935	0.600674	1.286533	0.103617	0.308634	0.614611	0.033565	0.680244
E4	0.046795	0.106403	0.750861	0	0.136364	1.350427	2.388278	0.057082	0.195559	0.414068	0.124711	1.941176
E5	0.423309	0.075361	0.677419	0.42562	0	0.251827	0.502372	0.105717	0.859504	10.1437	0.737027	0.284077
E6	0.389808	0.06082	0.663858	1.421652	0.133041	0	1.607184	0.040864	0.203913	0.36541	0.115076	2.303571
E7	1.058824	0.574528	1.312321	2.800366	0.186083	1.607184	0	0.665391	0.264431	0.531753	0.423063	2.04414
E8	2.218605	4.376068	0.31821	0.479915	0.239123	0.252891	0.875957	0	0.195505	0.342624	0.202422	0.22164
E9	0.379763	0.032423	0.682529	0.430401	5.834711	0.269752	0.498076	0.377096	0	4.206074	1.285374	0.284666
E10	0.435407	0.166612	0.64108	0.359681	4.131965	0.243498	0.699463	0.189667	2.904555	0	0.02503	0.27222
E11	0.394745	0.019726	0.014327	0.124711	0.077397	0.086233	0.037091	0.038062	0.093005	0.01907	0	0.105011
E12	0.360294	0.366086	0.731161	2.063725	0.15567	2.928571	2.082192	0.383072	0.243883	0.376786	0.133099	0

Table 11 - Equation of the Bestlines used in CBG (y=mx + b)

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
T1	0	1.66E-5 + .06	1.25E-5 + .08	1.55E-5 + 0	1.55E-5 + .01	1.71E-5 + .005	1.33E-5 + .08	1.43E-5 + .077	1.55E-5 + 0	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T2	1.25E-5 + .08	0	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + .01	1.71E-5 + .005	1.33E-5 + .08	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T3	1.25E-5 + .08	1.66E-5 + .06	0	1.53E-5 + .02	1.55E-5 + .01	1.71E-5 + .005	1.55E-5 + 0	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T4	1.55E-5 + 0	1.66E-5 + .06	1.25E-5 + .08	0	1.55E-5 + .01	1.71E-5 + .005	1.33E-5 + .08	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T5	1.25E-5 + .08	1.66E-5 + .06	1.25E-5 + .08	1.53E-5 + .02	0	1.71E-5 + .005	1.33E-5 + .08	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T6	1.25E-5 + .08	1.66E-5 + .06	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + .01	0	1.33E-5 + .08	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T7	1.25E-5 + .08	1.66E-5 + .06	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + .01	1.71E-5 + .005	0	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	1.55E-5 + 0
T8	1.55E-5 + 0	1.55E-5 + 0	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + 0	1.71E-5 + .005	1.33E-5 + .08	0	1.55E-5 + 0	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T9	1.25E-5 + .08	1.66E-5 + .06	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + 0	1.71E-5 + .005	1.33E-5 + .08	1.55E-5 + 0	0	1.55E-5 + 0	1.65E-5 + 0	1.73E-5 + .002
T10	1.25E-5 + .08	1.66E-5 + .06	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + .01	1.71E-5 + .005	1.33E-5 + .08	1.43E-5 + .077	1.54E-5 + .01	0	1.55E-5 + 0	1.73E-5 + .002
T11	1.25E-5 + .08	1.66E-5 + .06	1.55E-5 + 0	1.53E-5 + .02	1.55E-5 + .01	1.55E-5 + 0	1.55E-5 + 0	1.43E-5 + .077	1.54E-5 + .01	1.55E-5 + 0	0	1.55E-5 + 0
T12	1.25E-5 + .08	1.55E-5 + 0	1.25E-5 + .08	1.53E-5 + .02	1.55E-5 + .01	1.71E-5 + .005	1.33E-5 + .08	1.55E-5 + 0	1.54E-5 + .01	1.55E-5 + 0	1.65E-5 + 0	0

Table 12 - CBG Results (Area)

Target Node	Area (mi^2)
Boston (T1)	57,925
Hartford (T2)	25,785
Houston (T3).	646,299
Jacksonville (T4)	42,1753
Los Angeles (T5)	90,477
Miami (T6)	1,668,759
New Orleans (T7)	1,141,937
New York (T8)	583,191
San Diego (T9)	774,294
San Jose (T10)	1,284,112
Seattle (T11)	508,347
Tampa (T12)	1,052,078

Figure 12. CBG Results for T2 (Hartford)

Table 13 - Summary of CBG and UBM results

	CBG	UBM
Mean Miss	661.11 miles	770.69 miles
Percent Error	78.4%	99.6%
Mean Area	687,913 mi2	1,315,534 mi2

4.3 TTLH

The TTLH methodology was used to identify the nearest node using the collected delay measurements. A computer program enumerated all possible combinations of target nodes, polling nodes, and end nodes. In each trial, the maximum numbers of end nodes were used. The results show a trend in which the accuracy of the data increases slightly when the number of polling nodes increases. In this research, the number of end nodes dropped as the polling nodes increased. When nine polling nodes were used, TTLH only had two end nodes to choose from as the nearest node. The results are shown in Figure 13.

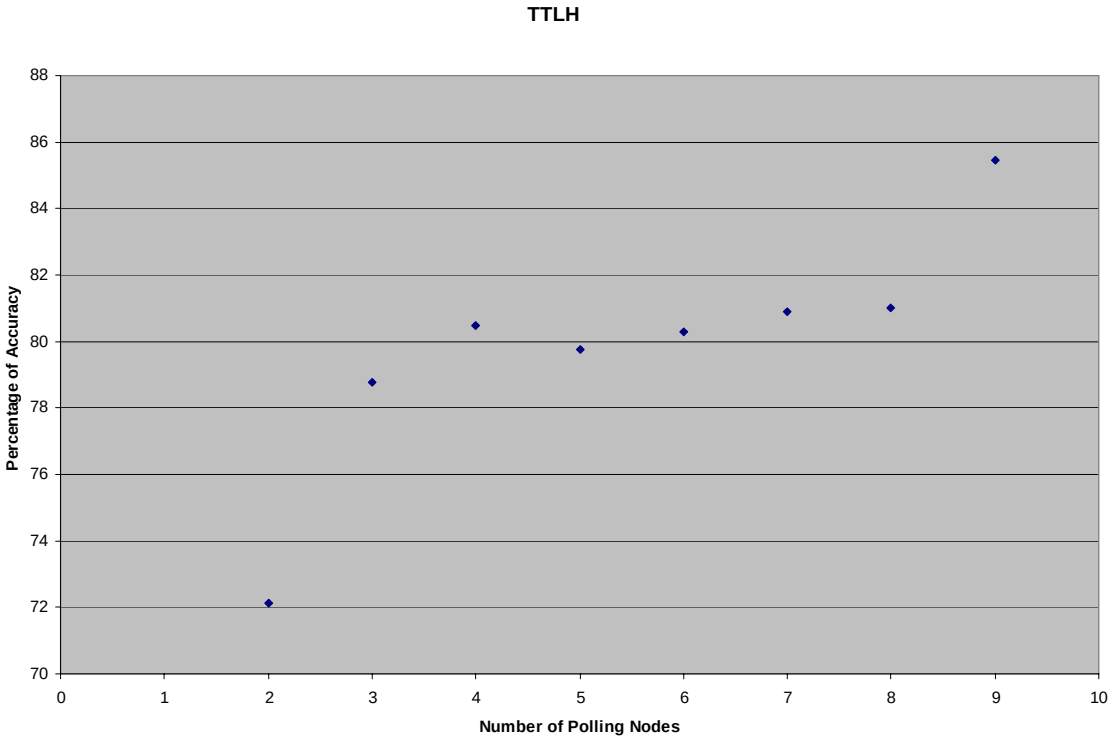

Figure 13. TTLH Scatterplot of Polling Nodes and Accuracy

A couple of interesting trends were discovered when the results were analyzed. The TTLH failed to identify the nearest-known node to Boston (T1) only when the New York (T8) and Hartford (T2) nodes were used as polling nodes, not as end nodes. When the target node Boston had a choice of either Hartford or New York as an end node, TTLH was 100% successful.

The three nodes that were most successful overall were the nodes located in Boston (T1), Hartford (T2), and New York (T8). No other three nodes in the polling set have nodes that are located closer to each other than these three. Most of the failures to identify the nearest node to any of these three nodes happen when neither of the other two nodes are used as end nodes. The accuracy rates from each of the eight ways to organize the TTLH test are shown in the Tables 14 - 21:

Table 14 - TTLH Results 2 Polling Nodes and 9 End Nodes

2 Polling Nodes, 9 End Nodes			
	Good	Bad	Rate
T1	53	2	0.9636
T2	50	5	0.9091
T3	50	5	0.9091
T4	48	7	0.8727
T5	17	38	0.3091
T6	47	8	0.8545
T7	49	6	0.8909
T8	49	6	0.8909
T9	18	37	0.3273
T10	28	27	0.5091
T11	19	36	0.3455
T12	48	7	0.8727
Overall	476	184	0.7212

Table 15 - TTLH Results 3 Polling Nodes and 8 End Nodes

3 Polling Nodes, 8 End Nodes			
	Good	Bad	Rate
T1	157	8	0.9515
T2	161	4	0.9758
T3	152	13	0.9212
T4	152	13	0.9212
T5	81	84	0.4909
T6	150	15	0.9091
T7	138	27	0.8364
T8	161	4	0.9758
T9	81	84	0.4909
T10	94	71	0.5697
T11	81	84	0.4909
T12	152	13	0.9212
Overall	1560	420	0.7879

Table 16 - TTLH Results 4 Polling Nodes and 7 End Nodes

4 Polling Nodes, 7 End Nodes			
	Good	Bad	Rate
T1	308	22	0.9333
T2	327	3	0.9909
T3	289	41	0.8758
T4	294	36	0.8909
T5	204	126	0.6182
T6	278	52	0.8424
T7	238	92	0.7212
T8	329	1	0.9970
T9	204	126	0.6182
T10	218	112	0.6606
T11	204	126	0.6182
T12	294	36	0.8909
Overall	3187	773	0.8048

Table 17 - TTLH Results 5 Polling Nodes and 6 End Nodes

5 Polling Nodes, 6 End Nodes			
	Good	Bad	Rate
T1	422	40	0.9134
T2	448	14	0.9697
T3	418	44	0.9048
T4	385	77	0.8333
T5	335	127	0.7251
T6	368	94	0.7965
T7	364	98	0.7879
T8	460	2	0.9957
T9	335	127	0.7251
T10	306	156	0.6623
T11	196	266	0.4242
T12	385	77	0.8333
Overall	4422	1122	0.7976

Table 18 - TTLH Results 6 Polling Nodes and 5 End Nodes

6 Polling Nodes, 5 End Nodes			
	Good	Bad	Rate
T1	421	41	0.9113
T2	438	24	0.9481
T3	414	48	0.8961
T4	359	103	0.7771
T5	356	106	0.7706
T6	353	109	0.7641
T7	346	116	0.7489
T8	454	8	0.9827
T9	365	97	0.7900
T10	324	138	0.7013
T11	262	200	0.5671
T12	359	103	0.7771
Overall	4451	1093	0.8028

Table 19 - TTLH Results 7 Polling Nodes and 4 End Nodes

7 Polling Nodes, 4 End Nodes			
	Good	Bad	Rate
T1	302	28	0.9152
T2	308	22	0.9333
T3	292	38	0.8848
T4	247	83	0.7485
T5	270	60	0.8182
T6	246	84	0.7455
T7	245	85	0.7424
T8	317	13	0.9606
T9	273	57	0.8273
T10	244	86	0.7394
T11	212	118	0.6424
T12	247	83	0.7485
Overall	3203	757	0.8088

Table 20 - TTLH Results 8 Polling Nodes and 3 End Nodes

8 Polling Nodes, 3 End Nodes			
	Good	Bad	Rate
T1	150	15	0.9091
T2	151	14	0.9152
T3	140	25	0.8485
T4	126	39	0.7636
T5	130	35	0.7879
T6	122	43	0.7394
T7	134	31	0.8121
T8	153	12	0.9273
T9	131	34	0.7939
T10	123	42	0.7455
T11	118	47	0.7152
T12	126	39	0.7636
Overall	1604	376	0.8101

Table 21 - TTLH Results 9 Polling Nodes and 2 End Nodes

9 Polling Nodes, 2 End Nodes			
	Good	Bad	Rate
T1	52	3	0.9455
T2	52	3	0.9455
T3	48	7	0.8727
T4	47	8	0.8545
T5	45	10	0.8182
T6	46	9	0.8364
T7	52	3	0.9455
T8	52	3	0.9455
T9	45	10	0.8182
T10	42	13	0.7636
T11	36	19	0.6545
T12	47	8	0.8545
Overall	564	96	0.8545

4.3.1 TTLH Results

The results from all 24,288 combinations correctly identified the nearest-known node at a rate of 80.15%. The nodes with the lowest accuracy rates were located on the West coast. TTLH uses the Euclidean distance formula to measure network distance. The network distance is assumed to correspond to a physical distance. An analysis of the simulated network topology revealed that the network distances did not always directly correspond to a short physical distance. In most cases, the nearest-known node using network measurements will also be the nearest known-node physically. The West coast nodes fail more often then the other nodes because of a flaw in the network design. The design on the West coast does not allow for the shortest network distance to be the shortest physical distance in all cases.

When analyzing the examples that failed to correctly identify the nearest node, a couple of trends were noticed. The TTLH results failed more often when the actual

nearest nodes were utilized as polling nodes. The correct nearest node was identified more often when the actual nearest node was used as an end node.

To test the assumption that the network topology can influence the results of TTLH, a thirteenth node was introduced into the project. This node was connected so that the closest network node would not be the geographically closest node. The thirteenth node was located in Birmingham. Its closest node on the network was the Jacksonville node, but the shortest physical node was in New Orleans. As expected, TTLH consistently identified the nearest-known node as the nearest node on the network. Since the nearest geographic node was not correctly identified those results were considered incorrect. When the Birmingham node was introduced, the overall accuracy of TTLH dropped from 80.15% down to 64.07%. Having an understanding of the network topology will help the researcher know when TTLH is providing the nearest-known physical node incorrectly.

4.4 Hybrid Methodology

The results of the TTLH show that the positioning of nodes has an effect on the results. An objective of this research was to determine if the accuracy of the delay-based IP Geolocation methods could be improved. For this reason, a hybrid methodology is proposed which uses a combination of UBM, CBG and TTLH when trying to geolocate a device by only using delay measurements. The researcher used the results of the UBM and CBG examples to choose the location of the end nodes and polling nodes used for the TTLH example. The TTLH examples show the best results when it has end nodes located closer to the target than polling nodes. UBM and CBG can be used to give an approximate location of the target. This was calculated for all twelve targets, but for the ease of discussion one target was selected to show how this hybrid methodology will work.

The first step in this methodology would be to use UBM to determine the wide area a target is located. This is done by finding one-way delay times to a target node and estimating the upper bound by using 2/3 speed of light for rate. Once those distances are known they can be drawn to a map and triangulation can be used to determine the approximate area where the target is located. This was accomplished for each of the targets. An example from the target located in Hartford is shown in Figure 10. That resulting area immediately tells us that the target node is located somewhere in the northern part of the east coast.

The next step used constraint-based geolocation to find a more accurate rate of transmission. The bestline was determined from each polling node. The delay times

were plotted on the bestline, and x intercepts were used to determine the distance from the polling node to the target. Those distances were then drawn on a map. Multilateration gives an area of overlap that contains the target.

Now that two areas have been found for the location of the target, the CBG results can be compared with the UBM results to determine if they agree on the location of the target. Then TTLH can be accomplished by first selecting end nodes that are located in or near the area of overlap. In this example, end node 1 and end node 8 are the only two end nodes that are located close to the shaded area from the CBG. TTLH was conducted using the combination of two end nodes and nine polling nodes. That combination was chosen because only two end nodes are located near the shaded area. The greatest number of polling nodes available was nine. The final step was to find the Euclidean distances of the two selected end nodes with the maximum number of available polling nodes.

4.4.1 Hybrid Methodology Results

End node 1 resulted in the lowest Euclidean distance, so it was determined that the target node is located closest to that node. The target node selected for this example, is located in Hartford. The end node that was selected as the nearest node is located in Boston. The results were correct; Boston is the nearest known node to Hartford in this simulation.

That methodology was repeated for the other eleven nodes in this example. Using the hybrid methodology, end nodes were selected for TTLH based on their proximity to the overlapping areas from constraint-based geolocation. In eleven of the twelve cases,

TTLH returned the correct nearest node when the end nodes were located closest to the target. The one case that did not identify the nearest geographic node did correctly identify the nearest network node. That one case where the hybrid methodology failed was when Seattle was the target node. The hybrid results claimed Los Angeles to be the nearest node, but San Jose is in fact the correct nearest node. A flaw in the network design is to blame for this result; the network paths to Los Angeles are closer in length to the paths to Seattle than the network paths to the correct nearest node San Jose. The underlying assumption to TTLH is that the network distance corresponds to the physical distance. It was discussed earlier in this paper that networks often follow shortest routes to the destination but is not always the case.

Using the hybrid methodology, the success rate of TTLH improved. The shotgun approach to TTLH yielded an overall success rate of 80.15%, but when the hybrid methodology was used the success rate increased to 91.66%.

4.5 Summary

The data collected and presented in this chapter showed that the CBG is a more accurate than the UBM method for geolocation. Both of these methodologies showed that the location of a target node can be found to some levels of granularity. The TTLH results showed that the positioning of the nodes has an effect on the results of the test. The accuracy rates of TTLH improve when an end node is closely located to the target node. Based on the results, a hybrid methodology was presented and tested. The hybrid method uses the results of the CBG method to select end nodes that are most likely to be located close to the target. When the hybrid methodology was tested, the accuracy rates of finding the nearest-known node improved.

5. Conclusions and Recommendations

The ability to determine the geographic location of a node on the Internet based upon its Internet Protocol address is an essential tool for many commercial and military applications. IP Geolocation can also be used to add another layer of protection. When authenticating users, locate the source of connection attempts to sensitive information assets, and to locate unmapped nodes. While the current methods used for IP Geolocation are not exact, in many cases they provide an acceptable estimate of the physical location of the IP address.

Geolocation can be accomplished through a number of methodologies. Those methodologies fall into three categories. 1) methods that store location information in databases, 2) methods that use information leakage to find the target locations, and 3) methods that calculate the location based upon delay measurements. This project focused on the third method, because collecting delay measurements give complete control of the accuracy of the data to the researcher. The information used in the other methods is easily corruptible, either purposely or inadvertently. The methods that were analyzed in a simulation environment for this project found the location of the target node through the use of multilateration and the nearest known node.

In this thesis, various methods for IP Geolocation were introduced, a brief understanding of their operation was provided, and the research investigation of delay-based methods was discussed. Analysis of the collected results show that distance estimations could be made from delay measurements. The UBM and CBG methods use delay measurements to determine the upper bound on the distance to the target location.

Replicating these geolocation methods found that they can be used to determine the geographic region of a target to some granularity. This project also showed that CBG is a more accurate method than UBM.

The TTLH was selected as a nearest-known node methodology to replicate. This methodology does not return the area of the target, but rather the known node that is located closest to the target. Since the calculations are all done with network data, an assumption of the nearest known node methodologies is that the nearest known node on the network is also the nearest known node physically. This assumption was tested by adding a node on the network where the nearest network node was not the nearest physical node. The accuracy rates of TTLH dropped dramatically after that node was included. The network infrastructure can cause false geolocation results.

A hybrid methodology was proposed that uses the results of multilateration geolocation to select nodes to be used for nearest-known node experiments. Perhaps the most important finding of this project was that the results from the nearest-known node experiments showed that the positioning of the end nodes in relation to the target had an effect on the accuracy of the results. The hybrid methodology first finds the area that a target is located by conducting the multilateration methodologies. Then it selects end nodes that are located in or close to that area. That hybrid methodology was tested and it was found that the accuracy of the finding the nearest-known node was increased.

5.1 Summary of Results

The UBM results were promising. Art first glance they do not look very accurate. On average the estimations were overshooting the target by a factor of two, and the

resulting mean area of overlap for all targets was 1,315,534 square miles which is five times larger then the state of Texas. Looking closely at the results showed many target locations that were much smaller then the mean and when those area were plotted, much of the area fell over the ocean. The resulting area for each target always would exclude a portion of the country. The very pessimistic constant rate of 2/3 speed of light did reveal the target's location to some granularity. That can be used in the overall geolocation effort that should include more then one methodology.

The CBG results were more accurate than the UBM results. This was not unexpected because the CBG uses known network nodes to find a more accurate rate of transmission than the UBM. The mean area for all targets in CBG was about half the size of the UBM results at 687,913 square miles. The most interesting finding from analyzing the CBG data was that the method strives to ensure the target's location is included in the resulting area. The bestline is defaults to a pessimistic rate by using the line that is close to but below all data points. The bestline could use a more accurate rate by using the mean of all the data points, but that would then underestimate the distance to the target, thus not including the target in the overlapping area. By defaulting to pessimistic rates CBG method sacrifices accuracy to ensure the targets location is including in the resulting area.

The TTLH data also showed some interesting findings. The overall accuracy rate of 80.15% gave plenty of examples to study to find when the method would fail to return the closest physical node to the target node. Although the closest physical node was only determined on average 4 out of 5 times, the closest network node was always returned correctly. The TTLH assumes that the closest network node is also the closest physical

node. Most often that is the case. The need for some knowledge of the network infrastructure for the target is beneficial to someone using TTLH for geolocation.

The TTLH results led to the development of a new hybrid methodology. TTLH was correctly returning the nearest network node, but that was not always the nearest physical node. It also was more accurate when the target had end nodes closer to it then polling nodes. By using the resulting areas from UBM and CBG one can select end nodes that are geographically close to the target. Then the TTLH method is conducted with those selected end nodes. This hybrid method was conducted for all twelve targets used in this simulation. The correct nearest node was returned eleven times for an accuracy rate of 91.66%. The one target node that was not correctly identified was the Seattle node. Having an idea of the network topology made it easy to see why San Jose was not selected as the nearest node to Seattle. The node in Los Angeles was closer on the network to Seattle than San Jose. Knowing the network infrastructure can help determine when TTLH is returning incorrect results. Using the hybrid methodology that was introduced here in section 4.4 can help someone achieve more accurate geolocation results than when a single methodology is employed.

5.2 Significance of Research

The significance of the research is that the accuracy of delay-based IP Geolocation can be improved when combining methods together into a hybrid methodology. The results will enable the reader to apply the new hybrid methodology and improve real-world applications of delay-based IP Geolocation.

5.3 Limitations

This project was limited by the size of the network in the simulation. A more robust network would have allowed more network paths to travel in the shortest geographic distances. A larger network would also have allowed for more end nodes, polling nodes and target nodes to add to the complexity of the examples.

The number of nodes used in this data was twelve. This relatively small number limited the complexity of calculations that could be completed. As polling nodes are increased, multilateration increases the number of circles that are determining the area of the target. When a polling node is added that area has a possibility of shrinking.

TTLH was conducted using the maximum number of polling nodes for each test. As the number of polling nodes increased, the number of end nodes decreased. This may have limited the accuracy of TTLH because when nine polling nodes were used, it was a 50% of being right.

5.4 Future Research

The possibilities of future research are plentiful. This project introduced a methodology that would be interesting to test in a real world network. Combining the Upper-Bound Multilateration, Constraint- Based Geolocation, and TTLH into a real life experiment could yield interesting results. This could be done by replicating the hybrid methodology introduced here. Other geolocation methods such as whois could be used to increase the number and find the location of end nodes used in the experiment.

It would also be beneficial to change the scale of this research project to a smaller scale such as the size of a state or a city. The granularity of the accuracy of the results is

dependant upon the numbers and the locations of the end nodes. Improving the granularity of the geolocation results would be a great benefit. Another possible area of research would be to translate these methods to a real life experiment. The TTLH could be tested in a MAN by using target, end, and polling nodes located around a metropolitan area that are under the control of the researcher.

It would be interesting to study the TTLH methodology to find at what point adding polling nodes does not improve the accuracy of the data. Further, it is desired to conduct a detailed analysis of the impact of varying line speeds to selected nodes with the goal of developing a correction factor in the Euclidian distance formula.

APPENDICIES

Appendix A: CollectIt12.c “C” Program

Appendix B: Area12.c “C” Program

Appendix A: CollectIt12.c “C” Program

```
/* begin CollectIt12.c */

/* A program to analyze IP Geolocation data for 12 nodes */

/* includes */
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/* defines */
#define DEBUG_MODE 0
#define NOT_QUIET 0
#define TOTAL_NODES ((int) 12)

/* structures */
struct IPGCase
{
   int TN;
   int TI;
   int NPN;
   int PNIL[TOTAL_NODES];
   int NEN;
   int ENIL[TOTAL_NODES];
   double ED[TOTAL_NODES];
   int MED;
};

/* function prototypes */
int CalculateEuclidianDistance(double *, int, int *, int, int *, int);

/* globals */
double Delays[TOTAL_NODES][TOTAL_NODES];
char PollingNodes[TOTAL_NODES+1][100];

/* functions */
int CalculateEuclidianDistance(double *Euclidian, int NumPNs, int *PNI, int NumENs, int *ENI, int TI)
{
   int i;
   int j;
   int row1;
   int col1;
```

```
   int row2;
   int col2;
   double Accumulator;
   double Term;

   Accumulator = (double) 0.0;
   Term = (double) 0.0;

#if 0
   for(i=0; i<13; i++)
   {
      for(j=0; j<13; j++)
          {
         printf("Delays[%d][%d] = %f\n", i, j, Delays[i][j]);
          }
   }
   printf("You have %d Polling Nodes\n", NumPNs);
   for(i=0; i<NumPNs; i++)
   {
          printf("PN %d is %d\n", i, PNI[i]);
   }
   printf("You have %d End Nodes\n", NumENs);
   for(i=0; i<NumENs; i++)
   {
          printf("EN %d is %d\n", i, ENI[i]);
   }
   printf("Target Node is %d\n", TI);
#endif

   for(i=0; i<NumENs; i++)
   {
         Accumulator = (double) 0.0;
      Term = (double) 0.0;
      for(j=0; j<NumPNs; j++)
          {
         row1 = ENI[i];
             col1 = PNI[j];
         row2 = TI;
         col2 = PNI[j];

                Term = Delays[row1][col1] - Delays[row2][col2];

                /*printf("Term %d %d is %f\n", i, j, Term);*/

                Term = Term * Term;
             Accumulator += Term;
          }
```

```
      Euclidian[i] = (double) sqrt(Accumulator);
   }


   return(0);
}

int main(int argc, char* argv[])
{
   char InputLine[5000];
   char Token[20][100];
   char PN[TOTAL_NODES+1];
   char TN[TOTAL_NODES+1];
   char Selected[TOTAL_NODES];
   char GoodFilename[100];
   char BadFilename[100];
   unsigned long combinations;
   unsigned long GoodCount[TOTAL_NODES];
   unsigned long BadCount[TOTAL_NODES];
   int result;
   int p1, p2, p3, p4, p5, p6, p7, p8, p9, p10;
   int e1, e2, e3, e4, e5, e6, e7, e8, e9, e10;
   int i;
   int j;
   int c;
   int Length;
   int Tokens;
   int LineCount;
   int First;
   int TotalNodes;
   int NumberPollingNodes;
   int CurrentPollingNodes;
   int PollingNodeIndexList[TOTAL_NODES];
   int NumberEndNodes;
   int EndNodeIndexList[TOTAL_NODES];
   int TargetNodeIndex;
   int minimumindex;
   int minimumCFindex;
   int minimumDindex;
   int done;
   double CrowFlies[TOTAL_NODES][TOTAL_NODES];
   double DrivingDistance[TOTAL_NODES][TOTAL_NODES];
   double EuclidianDistance[TOTAL_NODES];
   double CFT[13];
   double DT[13];
   double minimum;
   double minimumCF;
```

```
double minimumD;
FILE *goodfile;
FILE *badfile;
FILE *outfile;
FILE *costfile;
struct IPGCase TenLargestMagnitudes[10];
struct IPGCase TenSmallestMagnitudes[10];

printf("Version 03 March 2007\n");

/* initialize Largest and Smallest structures */
for(i=0; i<10; i++)
{
   TenLargestMagnitudes[i].TN=TOTAL_NODES;
   TenLargestMagnitudes[i].TI=0;
   TenLargestMagnitudes[i].NPN=0;
   TenLargestMagnitudes[i].NEN=0;
   TenLargestMagnitudes[i].MED=0;
   for(j=0; j<TOTAL_NODES; j++)
   {
      TenLargestMagnitudes[i].PNIL[j]=0;
      TenLargestMagnitudes[i].ENIL[j]=0;
      TenLargestMagnitudes[i].ED[j]=(double)0.0;
   }

   TenSmallestMagnitudes[i].TN=TOTAL_NODES;
   TenSmallestMagnitudes[i].TI=0;
   TenSmallestMagnitudes[i].NPN=0;
   TenSmallestMagnitudes[i].NEN=0;
   TenSmallestMagnitudes[i].MED=0;
   for(j=0; j<TOTAL_NODES; j++)
   {
      TenSmallestMagnitudes[i].PNIL[j]=0;
      TenSmallestMagnitudes[i].ENIL[j]=0;
      TenSmallestMagnitudes[i].ED[j]=(double)1000000.0;
   }
}

printf("Opening out.csv for write...\n");
/*costfile=fopen("d:\\out.csv","w");*/
outfile=fopen("./out.csv","w");
if(outfile==NULL)
{
   printf("\n Error cannot open file out.csv\a ");
   exit(0);
}
```

```
printf("Reading DelayData.csv file...\n");
/*costfile=fopen("d:\\DelayData.csv","r");*/
costfile=fopen("./DelayData.csv","r");
if(costfile==NULL)
{
   printf("\n Error cannot open file DelayData.csv\a ");
   exit(0);
}
LineCount = 0;
First = 1;
while(fgets(InputLine,4999,costfile) != NULL)
{
   /* get a line */
   LineCount++;
   Length = strlen(InputLine);
   /* tokenize it */
   Tokens=0;
   j=0;
   for(i=0; i<Length; i++)
   {
      if(InputLine[i] == 44)
      {
         /* delimiter */
         Token[Tokens][j] = 0;
         j = 0;
         Tokens++;
      }
      else
      {
         /* character */
         Token[Tokens][j++] = InputLine[i];
      }
   }
   Token[Tokens++][j] = 0;
   /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
   if(First)
   {
      for(i=0; i<Tokens; i++)
      {
         strcpy(PollingNodes[i],&Token[i][0]);
          /*sprintf(PollingNodes[i], "%s", Token[i][0]);*/
      }
      First=0;
   }
   else
   {
```

```
         /* data */
         for(i=0; i<Tokens; i++)
         {
            if(i)
            {
               Delays[LineCount-2][i-1] = (double)atof(Token[i]);
            }
            /*sprintf(PollingNodes[i], "%s", Token[i][0]);*/
         }
      }
#if DEBUG_MODE
      printf("Line %d with %d tokens\n",LineCount,Tokens);
      for(i=0; i<Tokens; i++)
      {
         printf("Token %d is %s\n",i,&Token[i][0]);
      }
#endif
      InputLine[Length-1]=0;
      fprintf(outfile, "%s\n", InputLine);
   }
   printf("Closing DelayData.csv file...\n");
   fclose(costfile);

   printf("Reading CrowFlies.csv file...\n");
   /*costfile=fopen("d:\\CrowFlies.csv","r");*/
   costfile=fopen("./CrowFlies.csv","r");
   if(costfile==NULL)
   {
      printf("\n Error cannot open file CrowFlies.csv\a ");
      exit(0);
   }
   LineCount = 0;
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* comma delimiter */
             Token[Tokens][j] = 0;
```

```
            j = 0;
            Tokens++;
          }
          else
          {
             /* character */
             Token[Tokens][j++] = InputLine[i];
          }
       }
       Token[Tokens++][j] = 0;
       /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
       if(First)
       {
          First=0;
       }
       else
       {
          /* data */
          for(i=0; i<Tokens; i++)
          {
              if(i)
              {
                 CrowFlies[LineCount-2][i-1] = (double)atof(Token[i]);
              }
          }
        }
#if DEBUG_MODE
        printf("Line %d with %d tokens\n",LineCount,Tokens);
        for(i=0; i<Tokens; i++)
        {
           printf("Token %d is %s\n",i,&Token[i][0]);
        }
#endif
        InputLine[Length-1]=0;
    }
    printf("Closing CrowFlies.csv file...\n");
    fclose(costfile);

    printf("Reading Driving.csv file...\n");
    /*costfile=fopen("d:\\Driving.csv","r");*/
    costfile=fopen("./Driving.csv","r");
    if(costfile==NULL)
    {
       printf("\n Error cannot open file Driving.csv\a ");
       exit(0);
    }
    LineCount = 0;
```

```
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* comma delimiter */
            Token[Tokens][j] = 0;
            j = 0;
            Tokens++;
         }
         else
         {
            /* character */
            Token[Tokens][j++] = InputLine[i];
         }
      }
      Token[Tokens++][j] = 0;
      /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
      if(First)
      {
         First=0;
      }
      else
      {
         /* data */
         for(i=0; i<Tokens; i++)
         {
            if(i)
            {
               DrivingDistance[LineCount-2][i-1] = (double)atof(Token[i]);
            }
         }
      }
#if DEBUG_MODE
      printf("Line %d with %d tokens\n",LineCount,Tokens);
      for(i=0; i<Tokens; i++)
      {
         printf("Token %d is %s\n",i,&Token[i][0]);
      }
```

```
#endif
       InputLine[Length-1]=0;
   }
   printf("Closing Driving.csv file...\n");
   fclose(costfile);

   /* replace end of line carrige return with null termination for Tampa */
   PollingNodes[TOTAL_NODES][5]=0;

  /* generate combos */
  TotalNodes=12;

   for(CurrentPollingNodes=2; CurrentPollingNodes<(TOTAL_NODES-2); CurrentPollingNodes++)
   {
      NumberPollingNodes=CurrentPollingNodes;
      NumberEndNodes=TotalNodes-CurrentPollingNodes-1;
      printf("Starting Polling Nodes: %d End Nodes: %d\n", NumberPollingNodes, NumberEndNodes);

      combinations = (unsigned long)0;
      for(i=0; i<TOTAL_NODES; i++)
      {
         GoodCount[i] = (unsigned long)0;
         BadCount[i] = (unsigned long)0;
      }
      sprintf(GoodFilename,"GoodP%02dE%02d.txt",NumberPollingNodes,NumberEndNodes);
      sprintf(BadFilename,"BadP%02dE%02d.txt",NumberPollingNodes,NumberEndNodes);

      printf("Opening %s for write...\n",GoodFilename);
      /*goodfile=fopen("d:\\good.txt","w");*/
      goodfile=fopen(GoodFilename,"w");
      if(goodfile==NULL)
      {
         printf("\n Error cannot open file %s\n",GoodFilename);
         exit(0);
      }

      printf("Opening %s for write...\n",BadFilename);
      /*badfile=fopen("d:\\bad.txt","w");*/
      badfile=fopen(BadFilename,"w");
      if(badfile==NULL)
      {
         printf("\n Error cannot open file %s\n", BadFilename);
         exit(0);
      }

      for(i=0; i<TotalNodes; i++)
      {
```

```
/* select the target node */
TargetNodeIndex = i;
printf("Processing Target Node: %d\n",i);

switch(CurrentPollingNodes)
{
   case 2:
   /* Polling Nodes = 2 End Nodes = 9 */
    /* select polling nodes */
    for(p1=0; p1<TotalNodes; p1++)
    {
       for(p2=p1+1; p2<TotalNodes; p2++)
       {
                                 /* we have to check to insure not a target node */
                                 if( (p1 != i) && (p2 != i) )
                                {
                                    /* this combination is ok, so store it for later use */
                                    PollingNodeIndexList[0]=p1;
                                    PollingNodeIndexList[1]=p2;
                                    /* clear selected index */
                                    for(c=0; c<TotalNodes; c++)
                                    {
                                       Selected[c]=0;
                                    }
                                    Selected[i] = 1;
                                    Selected[p1] = 1;
                                    Selected[p2] = 1;
                                    /* select end nodes */
                                    for(e1=0; e1<TotalNodes; e1++)
                                    {
                                       for(e2=e1+1; e2<TotalNodes; e2++)
                                       {
                                          for(e3=e2+1; e3<TotalNodes; e3++)
                                          {
                                             for(e4=e3+1; e4<TotalNodes; e4++)
                                             {
                                                for(e5=e4+1; e5<TotalNodes; e5++)
                                                {
                                                   for(e6=e5+1; e6<TotalNodes; e6++)
                                                   {
                                                      for(e7=e6+1; e7<TotalNodes; e7++)
                                                      {
                                                         for(e8=e7+1; e8<TotalNodes; e8++)
                                                         {
                                                            for(e9=e8+1; e9<TotalNodes; e9++)
                                                            {
```

```
*/
!Selected[e4] && !Selected[e5] &&
!Selected[e9])
```

```
#if NOT_QUIET
",NumberPollingNodes);
",NumberEndNodes);
",NumberPollingNodes);
PollingNodes[PollingNodeIndexList[c]+1]);
",NumberEndNodes);
```

```
 /* this generates all possible combinations of End Nodes
 if(!Selected[e1] && !Selected[e2] && !Selected[e3] &&
    !Selected[e6] && !Selected[e7] && !Selected[e8] &&
{
   /* this combination is ok */
    combinations++;
  EndNodeIndexList[0]=e1;
  EndNodeIndexList[1]=e2;
  EndNodeIndexList[2]=e3;
  EndNodeIndexList[3]=e4;
  EndNodeIndexList[4]=e5;
  EndNodeIndexList[5]=e6;
  EndNodeIndexList[6]=e7;
  EndNodeIndexList[7]=e8;
  EndNodeIndexList[8]=e9;

    printf("\nTarget Node: %d ",i);
printf("Total Polling Nodes: %d Polling Nodes:

    for(c=0; c<NumberPollingNodes; c++)
    {
       printf("%d ", PollingNodeIndexList[c]+1);
    }
    printf("Total End Nodes: %d End Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
     printf("%d ",EndNodeIndexList[c]+1);
    }
    printf("Target Node: %12s\n",PollingNodes[i+1]);
  printf("Total Polling Nodes: %d\nPolling Nodes:

    for(c=0; c<NumberPollingNodes; c++)
    {
     printf("%s  ",

    }
    printf("\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
     printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
    }
   printf("\n");
```

```
#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */




#if NOT_QUIET


Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else



#endif
```

```
  result =

          NumberPollingNodes,
        &PollingNodeIndexList[0],
        NumberEndNodes,
        &EndNodeIndexList[0],
        TargetNodeIndex);
  /* calculate crow flys and driving distance between

  for(c=0; c<NumberEndNodes; c++)
  {
   CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
   DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
  }


  for(c=0; c<NumberEndNodes; c++)
  {
     printf("%16s E[%02d] is %12f CrowFlies: %12f

     PollingNodes[EndNodeIndexList[c]+1], c,

  }

  if(!(combinations%100))
 {
    /*printf(".");*/
  }

  /* identify minimum euclidian entry */
  minimum=(double)1000000.0;
  for(c=0; c<NumberEndNodes; c++)
 {
    if(EuclidianDistance[c] < minimum)
    {
       minimumindex=c;
         minimum=EuclidianDistance[c];
    }
 }
  /* identify minimum crow flies entry */
  minimumCF=(double)1000000.0;
  for(c=0; c<NumberEndNodes; c++)
{
 if(CFT[c] < minimumCF)
 {
```

```
                                        minimumCFindex=c;
                                             minimumCF=CFT[c];
                                      }
                                   }
                                        /* identify minimum driving entry */
                                        minimumD=(double)1000000.0;
                                        for(c=0; c<NumberEndNodes; c++)
                                   {
                                      if(DT[c] < minimumD)
                                      {
                                         minimumDindex=c;
                                              minimumD=DT[c];
                                      }
                                   }
#if NOT_QUIET
                                        printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

                                        printf("Minimum Driving:    %12s  DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

                                        printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif
                                        if(minimumindex == minimumDindex)
                                   {
#if NOT_QUIET
                                      printf("\n*** CORRECT *** Minimum Euclidian Matches

Minimum Driving Distance\n");
#endif
                                          GoodCount[i]++;

                                      fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

Matches Minimum Driving Distance\n");
                                          fprintf(goodfile,"Target Node:

%12s\n",PollingNodes[i+1]);

                                      fprintf(goodfile,"Total Polling Nodes: %d\nPolling

Nodes: ",NumberPollingNodes);
                                          for(c=0; c<NumberPollingNodes; c++)
                                          {
                                       fprintf(goodfile,"%s  ",

PollingNodes[PollingNodeIndexList[c]+1]);
                                          }
```

```
Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif



Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);
```

```
   fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

   for(c=0; c<NumberEndNodes; c++)
   {
 fprintf(goodfile,"%s

   }
fprintf(goodfile,"\n");
   for(c=0; c<NumberEndNodes; c++)
   {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

   }
   fprintf(goodfile,"Minimum Euclidian:  %12s



   fprintf(goodfile,"Minimum Driving:    %12s



   fprintf(goodfile,"Minimum Crow Flies: %12s



 }
 else
{

printf("\n*** WRONG *** Minimum Euclidian Does Not


    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
```

```
PollingNodes[PollingNodeIndexList[c]+1]);


",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Minimum Crow Flies Distance\n");
#endif
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
```

```
                                  fprintf(badfile,"%s  ",

                                    }
                                    fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

                                    for(c=0; c<NumberEndNodes; c++)
                                    {
                                  fprintf(badfile,"%s

                                    }
                                 fprintf(badfile,"\n");
                                    for(c=0; c<NumberEndNodes; c++)
                                    {
                                  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

                                           PollingNodes[EndNodeIndexList[c]+1], c,

                                    }
                                    fprintf(badfile,"Minimum Euclidian:  %12s



                                    fprintf(badfile,"Minimum Driving:    %12s



                                    fprintf(badfile,"Minimum Crow Flies: %12s


                                 }

                                 if(minimumindex == minimumCFindex)
                            {

                               printf("*** CORRECT *** Minimum Euclidian Matches


                            }
```

```
                }
             }
          }
       }
    }
 }
}
break;

case 3:
/* Polling Nodes = 3 End Nodes = 8 */
 /* select polling nodes */
 for(p1=0; p1<TotalNodes; p1++)
 {
    for(p2=p1+1; p2<TotalNodes; p2++)
    {
       for(p3=p2+1; p3<TotalNodes; p3++)
      {
                              /* we have to check to insure not a target node */
                             if( (p1 != i) && (p2 != i) && (p3 != i) )
                            {
                                /* this combination is ok, so store it for later use */
                                 PollingNodeIndexList[0]=p1;
                                 PollingNodeIndexList[1]=p2;
                                 PollingNodeIndexList[2]=p3;
                                 /* clear selected index */
                                for(c=0; c<TotalNodes; c++)
                                {
                                   Selected[c]=0;
                                }
                                 Selected[i] = 1;
                                Selected[p1] = 1;
                                Selected[p2] = 1;
                                Selected[p3] = 1;
                                /* select end nodes */
                                 for(e1=0; e1<TotalNodes; e1++)
                                 {
                                    for(e2=e1+1; e2<TotalNodes; e2++)
                                   {
                                      for(e3=e2+1; e3<TotalNodes; e3++)
                                      {
                                         for(e4=e3+1; e4<TotalNodes; e4++)
                                         {
                                            for(e5=e4+1; e5<TotalNodes; e5++)
                                            {
                                               for(e6=e5+1; e6<TotalNodes; e6++)
                                               {
```

```
*/

!Selected[e4] && !Selected[e5] &&









#if NOT_QUIET

",NumberPollingNodes);



",NumberEndNodes);



",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);
```

```
for(e7=e6+1; e7<TotalNodes; e7++)
{
   for(e8=e7+1; e8<TotalNodes; e8++)
   {
             /* this generates all possible combinations of End Nodes

             if(!Selected[e1] && !Selected[e2] && !Selected[e3] &&

                !Selected[e6] && !Selected[e7] && !Selected[e8])
            {
               /* this combination is ok */
                combinations++;
               EndNodeIndexList[0]=e1;
               EndNodeIndexList[1]=e2;
               EndNodeIndexList[2]=e3;
               EndNodeIndexList[3]=e4;
               EndNodeIndexList[4]=e5;
               EndNodeIndexList[5]=e6;
               EndNodeIndexList[6]=e7;
               EndNodeIndexList[7]=e8;

                printf("\nTarget Node: %d ",i);
             printf("Total Polling Nodes: %d Polling Nodes:

                for(c=0; c<NumberPollingNodes; c++)
                {
                   printf("%d ", PollingNodeIndexList[c]+1);
                }
                printf("Total End Nodes: %d End Nodes:

                for(c=0; c<NumberEndNodes; c++)
                {
                 printf("%d ",EndNodeIndexList[c]+1);
                }
                printf("Target Node: %12s\n",PollingNodes[i+1]);
               printf("Total Polling Nodes: %d\nPolling Nodes:

                for(c=0; c<NumberPollingNodes; c++)
                {
                 printf("%s  ",

                }
                printf("\nTotal End Nodes: %d\nEnd Nodes:

                for(c=0; c<NumberEndNodes; c++)
                {
                 printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
```

```
#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */




#if NOT_QUIET


Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else



#endif
```

```
 }
 printf("\n");


 result =

         NumberPollingNodes,
        &PollingNodeIndexList[0],
        NumberEndNodes,
        &EndNodeIndexList[0],
        TargetNodeIndex);
 /* calculate crow flys and driving distance between

 for(c=0; c<NumberEndNodes; c++)
 {
  CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
  DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
 }


 for(c=0; c<NumberEndNodes; c++)
 {
    printf("%16s E[%02d] is %12f CrowFlies: %12f

    PollingNodes[EndNodeIndexList[c]+1], c,

 }

 if(!(combinations%100))
{
  /*printf(".");*/
 }

 /* identify minimum euclidian entry */
 minimum=(double)1000000.0;
 for(c=0; c<NumberEndNodes; c++)
{
   if(EuclidianDistance[c] < minimum)
   {
      minimumindex=c;
        minimum=EuclidianDistance[c];
   }
}
 /* identify minimum crow flies entry */
 minimumCF=(double)1000000.0;
 for(c=0; c<NumberEndNodes; c++)
{
```

```
   if(CFT[c] < minimumCF)
   {
      minimumCFindex=c;
           minimumCF=CFT[c];
   }
}
     /* identify minimum driving entry */
     minimumD=(double)1000000.0;
     for(c=0; c<NumberEndNodes; c++)
{
   if(DT[c] < minimumD)
   {
      minimumDindex=c;
           minimumD=DT[c];
   }
}
#if NOT_QUIET
     printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

     printf("Minimum Driving:    %12s  DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

     printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif

     if(minimumindex == minimumDindex)
{
#if NOT_QUIET

   printf("\n*** CORRECT *** Minimum Euclidian Matches

Minimum Driving Distance\n");
#endif

       GoodCount[i]++;

   fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

Matches Minimum Driving Distance\n");

       fprintf(goodfile,"Target Node:

%12s\n",PollingNodes[i+1]);

   fprintf(goodfile,"Total Polling Nodes: %d\nPolling

Nodes: ",NumberPollingNodes);

       for(c=0; c<NumberPollingNodes; c++)
       {
    fprintf(goodfile,"%s  ",

PollingNodes[PollingNodeIndexList[c]+1]);
```

```
Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif



Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);
```

```
    }
    fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%s

    }
fprintf(goodfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(goodfile,"Minimum Euclidian:  %12s



    fprintf(goodfile,"Minimum Driving:    %12s



    fprintf(goodfile,"Minimum Crow Flies: %12s



 }
 else
{

printf("\n*** WRONG *** Minimum Euclidian Does Not


    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
```

```
PollingNodes[PollingNodeIndexList[c]+1]);


",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Minimum Crow Flies Distance\n");
#endif

                                        }
                                    }
                                }
                            }
                        }
```

```
  fprintf(badfile,"%s  ",

  }
  fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

  for(c=0; c<NumberEndNodes; c++)
  {
  fprintf(badfile,"%s

  }
 fprintf(badfile,"\n");
  for(c=0; c<NumberEndNodes; c++)
  {
  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

           PollingNodes[EndNodeIndexList[c]+1], c,

  }
  fprintf(badfile,"Minimum Euclidian:  %12s



  fprintf(badfile,"Minimum Driving:    %12s



  fprintf(badfile,"Minimum Crow Flies: %12s


 }

 if(minimumindex == minimumCFindex)
{

printf("*** CORRECT *** Minimum Euclidian Matches


}
```

```
                    }
                 }
              }
           }
        }
     }
  }
}
break;

case 4:
/* Polling Nodes = 4 End Nodes = 7 */
 /* select polling nodes */
 for(p1=0; p1<TotalNodes; p1++)
 {
    for(p2=p1+1; p2<TotalNodes; p2++)
    {
       for(p3=p2+1; p3<TotalNodes; p3++)
      {
          for(p4=p3+1; p4<TotalNodes; p4++)
         {
                              /* we have to check to insure not a target node */
                             if( (p1 != i) && (p2 != i) && (p3 != i) && (p4 != i) )
                             {
                                 /* this combination is ok, so store it for later use */
                                  PollingNodeIndexList[0]=p1;
                                  PollingNodeIndexList[1]=p2;
                                  PollingNodeIndexList[2]=p3;
                                  PollingNodeIndexList[3]=p4;
                                  /* clear selected index */
                                 for(c=0; c<TotalNodes; c++)
                                 {
                                    Selected[c]=0;
                                 }
                                  Selected[i] = 1;
                                 Selected[p1] = 1;
                                 Selected[p2] = 1;
                                 Selected[p3] = 1;
                                 Selected[p4] = 1;
                                 /* select end nodes */
                                  for(e1=0; e1<TotalNodes; e1++)
                                 {
                                     for(e2=e1+1; e2<TotalNodes; e2++)
                                    {
                                       for(e3=e2+1; e3<TotalNodes; e3++)
                                       {
                                          for(e4=e3+1; e4<TotalNodes; e4++)
```

```
*/

!Selected[e4] && !Selected[e5] &&









#if NOT_QUIET

",NumberPollingNodes);



",NumberEndNodes);




",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);
```

```
{
   for(e5=e4+1; e5<TotalNodes; e5++)
   {
      for(e6=e5+1; e6<TotalNodes; e6++)
      {
         for(e7=e6+1; e7<TotalNodes; e7++)
         {
                     /* this generates all possible combinations of End Nodes

                     if(!Selected[e1] && !Selected[e2] && !Selected[e3] &&

                        !Selected[e6] && !Selected[e7] )
                   {
                     /* this combination is ok */
                      combinations++;
                     EndNodeIndexList[0]=e1;
                     EndNodeIndexList[1]=e2;
                     EndNodeIndexList[2]=e3;
                     EndNodeIndexList[3]=e4;
                     EndNodeIndexList[4]=e5;
                     EndNodeIndexList[5]=e6;
                     EndNodeIndexList[6]=e7;

                      printf("\nTarget Node: %d ",i);
                  printf("Total Polling Nodes: %d Polling Nodes:

                      for(c=0; c<NumberPollingNodes; c++)
                      {
                         printf("%d ", PollingNodeIndexList[c]+1);
                      }
                      printf("Total End Nodes: %d End Nodes:

                      for(c=0; c<NumberEndNodes; c++)
                      {
                       printf("%d ",EndNodeIndexList[c]+1);
                      }
                      printf("Target Node: %12s\n",PollingNodes[i+1]);
                    printf("Total Polling Nodes: %d\nPolling Nodes:

                      for(c=0; c<NumberPollingNodes; c++)
                      {
                       printf("%s  ",

                      }
                      printf("\nTotal End Nodes: %d\nEnd Nodes:

                      for(c=0; c<NumberEndNodes; c++)
```

```
#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */




#if NOT_QUIET


Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else



#endif
```

```
 {
  printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
 }
printf("\n");


 result =

         NumberPollingNodes,
        &PollingNodeIndexList[0],
        NumberEndNodes,
        &EndNodeIndexList[0],
        TargetNodeIndex);
 /* calculate crow flys and driving distance between

 for(c=0; c<NumberEndNodes; c++)
 {
  CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
  DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
 }


 for(c=0; c<NumberEndNodes; c++)
 {
    printf("%16s E[%02d] is %12f CrowFlies: %12f

    PollingNodes[EndNodeIndexList[c]+1], c,

 }

 if(!(combinations%100))
{
   /*printf(".");*/
 }

 /* identify minimum euclidian entry */
 minimum=(double)1000000.0;
 for(c=0; c<NumberEndNodes; c++)
{
   if(EuclidianDistance[c] < minimum)
   {
      minimumindex=c;
        minimum=EuclidianDistance[c];
   }
}
 /* identify minimum crow flies entry */
 minimumCF=(double)1000000.0;
```

```
    for(c=0; c<NumberEndNodes; c++)
{
   if(CFT[c] < minimumCF)
   {
      minimumCFindex=c;
           minimumCF=CFT[c];
   }
}
    /* identify minimum driving entry */
    minimumD=(double)1000000.0;
    for(c=0; c<NumberEndNodes; c++)
{
   if(DT[c] < minimumD)
   {
      minimumDindex=c;
           minimumD=DT[c];
   }
}

#if NOT_QUIET

    printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

    printf("Minimum Driving:    %12s  DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

    printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif

    if(minimumindex == minimumDindex)
{

#if NOT_QUIET

   printf("\n*** CORRECT *** Minimum Euclidian Matches

Minimum Driving Distance\n");
#endif

        GoodCount[i]++;

   fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

Matches Minimum Driving Distance\n");

        fprintf(goodfile,"Target Node:

%12s\n",PollingNodes[i+1]);

   fprintf(goodfile,"Total Polling Nodes: %d\nPolling

Nodes: ",NumberPollingNodes);

        for(c=0; c<NumberPollingNodes; c++)
        {
```

```
PollingNodes[PollingNodeIndexList[c]+1]);

Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);



%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);


#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif


Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);
```

```
 fprintf(goodfile,"%s  ",

   }
   fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

   for(c=0; c<NumberEndNodes; c++)
   {
 fprintf(goodfile,"%s

   }
fprintf(goodfile,"\n");
   for(c=0; c<NumberEndNodes; c++)
   {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

   }
   fprintf(goodfile,"Minimum Euclidian:  %12s



   fprintf(goodfile,"Minimum Driving:    %12s



   fprintf(goodfile,"Minimum Crow Flies: %12s



 }
 else

{

printf("\n*** WRONG *** Minimum Euclidian Does Not

    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling
```

```
PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);



%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Minimum Crow Flies Distance\n");
#endif

                                        }
                                    }
                                }
```

```
    for(c=0; c<NumberPollingNodes; c++)
    {
  fprintf(badfile,"%s  ",

    }
    fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%s

    }
 fprintf(badfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(badfile,"Minimum Euclidian:  %12s



    fprintf(badfile,"Minimum Driving:    %12s



    fprintf(badfile,"Minimum Crow Flies: %12s


 }

 if(minimumindex == minimumCFindex)
{

   printf("*** CORRECT *** Minimum Euclidian Matches


}
```

```
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
break;

case 5:
/* Polling Nodes = 5 End Nodes = 6 */
 /* select polling nodes */
 for(p1=0; p1<TotalNodes; p1++)
 {
    for(p2=p1+1; p2<TotalNodes; p2++)
    {
       for(p3=p2+1; p3<TotalNodes; p3++)
      {
          for(p4=p3+1; p4<TotalNodes; p4++)
         {
             for(p5=p4+1; p5<TotalNodes; p5++)
            {
                           /* we have to check to insure not a target node */
                          if( (p1 != i) && (p2 != i) && (p3 != i) && (p4 != i) && (p5 != i) )
                          {
                              /* this combination is ok, so store it for later use */
                              PollingNodeIndexList[0]=p1;
                              PollingNodeIndexList[1]=p2;
                              PollingNodeIndexList[2]=p3;
                              PollingNodeIndexList[3]=p4;
                              PollingNodeIndexList[4]=p5;
                              /* clear selected index */
                             for(c=0; c<TotalNodes; c++)
                             {
                                Selected[c]=0;
                             }
                              Selected[i] = 1;
                             Selected[p1] = 1;
                             Selected[p2] = 1;
                             Selected[p3] = 1;
                             Selected[p4] = 1;
                             Selected[p5] = 1;
                             /* select end nodes */
                              for(e1=0; e1<TotalNodes; e1++)
```

```
{
    for(e2=e1+1; e2<TotalNodes; e2++)
   {
      for(e3=e2+1; e3<TotalNodes; e3++)
      {
         for(e4=e3+1; e4<TotalNodes; e4++)
         {
            for(e5=e4+1; e5<TotalNodes; e5++)
            {
               for(e6=e5+1; e6<TotalNodes; e6++)
               {
                                 /* this generates all possible combinations of End Nodes

                                 if(!Selected[e1] && !Selected[e2] && !Selected[e3] &&

                                    !Selected[e6] )
                               {
                                  /* this combination is ok */
                                   combinations++;
                                  EndNodeIndexList[0]=e1;
                                  EndNodeIndexList[1]=e2;
                                  EndNodeIndexList[2]=e3;
                                  EndNodeIndexList[3]=e4;
                                  EndNodeIndexList[4]=e5;
                                  EndNodeIndexList[5]=e6;

                                    printf("\nTarget Node: %d ",i);
                              printf("Total Polling Nodes: %d Polling Nodes:

                                    for(c=0; c<NumberPollingNodes; c++)
                                    {
                                       printf("%d ", PollingNodeIndexList[c]+1);
                                    }
                                    printf("Total End Nodes: %d End Nodes:

                                    for(c=0; c<NumberEndNodes; c++)
                                    {
                                     printf("%d ",EndNodeIndexList[c]+1);
                                    }
                                    printf("Target Node: %12s\n",PollingNodes[i+1]);
                                  printf("Total Polling Nodes: %d\nPolling Nodes:

                                    for(c=0; c<NumberPollingNodes; c++)
                                    {
                                     printf("%s  ",

                                    }
```

```
*/

!Selected[e4] && !Selected[e5] &&
```

```
#if NOT_QUIET

",NumberPollingNodes);
```

```
",NumberEndNodes);
```

```
",NumberPollingNodes);

PollingNodes[PollingNodeIndexList[c]+1]);
```

```
",NumberEndNodes);



#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */



#if NOT_QUIET


Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else



#endif
```

```
 printf("\nTotal End Nodes: %d\nEnd Nodes:

 for(c=0; c<NumberEndNodes; c++)
 {
  printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
 }
printf("\n");


 result =

         NumberPollingNodes,
       &PollingNodeIndexList[0],
       NumberEndNodes,
       &EndNodeIndexList[0],
       TargetNodeIndex);
 /* calculate crow flys and driving distance between

 for(c=0; c<NumberEndNodes; c++)
 {
  CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
  DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
 }


 for(c=0; c<NumberEndNodes; c++)
 {
    printf("%16s E[%02d] is %12f CrowFlies: %12f

    PollingNodes[EndNodeIndexList[c]+1], c,

 }

 if(!(combinations%100))
{
   /*printf(".");*/
 }

 /* identify minimum euclidian entry */
 minimum=(double)1000000.0;
 for(c=0; c<NumberEndNodes; c++)
{
   if(EuclidianDistance[c] < minimum)
   {
      minimumindex=c;
        minimum=EuclidianDistance[c];
   }
```

```
                                                                    }
                                                                     /* identify minimum crow flies entry */
                                                                     minimumCF=(double)1000000.0;
                                                                     for(c=0; c<NumberEndNodes; c++)
                                                                {
                                                                    if(CFT[c] < minimumCF)
                                                                    {
                                                                       minimumCFindex=c;
                                                                            minimumCF=CFT[c];
                                                                    }
                                                                }
                                                                     /* identify minimum driving entry */
                                                                     minimumD=(double)1000000.0;
                                                                     for(c=0; c<NumberEndNodes; c++)
                                                                {
                                                                    if(DT[c] < minimumD)
                                                                    {
                                                                       minimumDindex=c;
                                                                            minimumD=DT[c];
                                                                    }
                                                                }
#if NOT_QUIET
                                                                     printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


                                                                     printf("Minimum Driving:    %12s  DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


                                                                     printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif

                                                                     if(minimumindex == minimumDindex)
                                                                {
#if NOT_QUIET
                                                                    printf("\n*** CORRECT *** Minimum Euclidian Matches

Minimum Driving Distance\n");
#endif
                                                                        GoodCount[i]++;

                                                                    fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

Matches Minimum Driving Distance\n");
                                                                        fprintf(goodfile,"Target Node:

%12s\n",PollingNodes[i+1]);
```

```
Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);


#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif


Not Matcs Minimum Driving Distance\n");
```

```
fprintf(goodfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
 fprintf(goodfile,"%s  ",

    }
    fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%s

    }
fprintf(goodfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(goodfile,"Minimum Euclidian:  %12s



    fprintf(goodfile,"Minimum Driving:    %12s



    fprintf(goodfile,"Minimum Crow Flies: %12s



 }
 else
{

printf("\n*** WRONG *** Minimum Euclidian Does Not


    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does
```

```
%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Minimum Crow Flies Distance\n");
#endif
```

```
    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
  fprintf(badfile,"%s  ",

    }
    fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%s

    }
 fprintf(badfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(badfile,"Minimum Euclidian:  %12s



    fprintf(badfile,"Minimum Driving:    %12s



    fprintf(badfile,"Minimum Crow Flies: %12s


 }

  if(minimumindex == minimumCFindex)
{

  printf("*** CORRECT *** Minimum Euclidian Matches


}
```

```
                                    }
                                 }
                              }
                           }
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
break;

case 6:
/* Polling Nodes = 6 End Nodes = 5 */
 /* select polling nodes */
 for(p1=0; p1<TotalNodes; p1++)
 {
    for(p2=p1+1; p2<TotalNodes; p2++)
    {
       for(p3=p2+1; p3<TotalNodes; p3++)
      {
          for(p4=p3+1; p4<TotalNodes; p4++)
         {
             for(p5=p4+1; p5<TotalNodes; p5++)
            {
                for(p6=p5+1; p6<TotalNodes; p6++)
               {
                              /* we have to check to insure not a target node */
                             if( (p1 != i) && (p2 != i) && (p3 != i) && (p4 != i) && (p5 != i) &&
                                 (p6 != i) )
                            {
                                /* this combination is ok, so store it for later use */
                                 PollingNodeIndexList[0]=p1;
                                 PollingNodeIndexList[1]=p2;
                                 PollingNodeIndexList[2]=p3;
                                 PollingNodeIndexList[3]=p4;
                                 PollingNodeIndexList[4]=p5;
                                 PollingNodeIndexList[5]=p6;
                                 /* clear selected index */
                                for(c=0; c<TotalNodes; c++)
                                {
                                   Selected[c]=0;
                                }
```

```
 Selected[i] = 1;
Selected[p1] = 1;
Selected[p2] = 1;
Selected[p3] = 1;
Selected[p4] = 1;
Selected[p5] = 1;
Selected[p6] = 1;
/* select end nodes */
 for(e1=0; e1<TotalNodes; e1++)
{
    for(e2=e1+1; e2<TotalNodes; e2++)
   {
      for(e3=e2+1; e3<TotalNodes; e3++)
      {
         for(e4=e3+1; e4<TotalNodes; e4++)
         {
            for(e5=e4+1; e5<TotalNodes; e5++)
            {
                                  /* this generates all possible combinations of End Nodes

                                  if(!Selected[e1] && !Selected[e2] && !Selected[e3] &&

                                 {
                                    /* this combination is ok */
                                     combinations++;
                                    EndNodeIndexList[0]=e1;
                                    EndNodeIndexList[1]=e2;
                                    EndNodeIndexList[2]=e3;
                                    EndNodeIndexList[3]=e4;
                                    EndNodeIndexList[4]=e5;

                                      printf("\nTarget Node: %d ",i);
                                 printf("Total Polling Nodes: %d Polling Nodes:

                                      for(c=0; c<NumberPollingNodes; c++)
                                      {
                                         printf("%d ", PollingNodeIndexList[c]+1);
                                      }
                                      printf("Total End Nodes: %d End Nodes:

                                      for(c=0; c<NumberEndNodes; c++)
                                      {
                                       printf("%d ",EndNodeIndexList[c]+1);
                                      }
                                      printf("Target Node: %12s\n",PollingNodes[i+1]);
                                    printf("Total Polling Nodes: %d\nPolling Nodes:
*/

!Selected[e4] && !Selected[e5] )




#if NOT_QUIET

",NumberPollingNodes);




",NumberEndNodes);




",NumberPollingNodes);
```

```
PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);



#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */



#if NOT_QUIET

Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else



#endif
```

```
 for(c=0; c<NumberPollingNodes; c++)
 {
  printf("%s  ",

 }
 printf("\nTotal End Nodes: %d\nEnd Nodes:

 for(c=0; c<NumberEndNodes; c++)
 {
  printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
 }
printf("\n");


 result =

         NumberPollingNodes,
       &PollingNodeIndexList[0],
       NumberEndNodes,
       &EndNodeIndexList[0],
       TargetNodeIndex);
 /* calculate crow flys and driving distance between

 for(c=0; c<NumberEndNodes; c++)
 {
  CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
  DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
 }


 for(c=0; c<NumberEndNodes; c++)
 {
    printf("%16s E[%02d] is %12f CrowFlies: %12f

    PollingNodes[EndNodeIndexList[c]+1], c,

 }

 if(!(combinations%100))
{
  /*printf(".");*/
 }

 /* identify minimum euclidian entry */
 minimum=(double)1000000.0;
 for(c=0; c<NumberEndNodes; c++)
{
```

```
      if(EuclidianDistance[c] < minimum)
      {
         minimumindex=c;
           minimum=EuclidianDistance[c];
      }
   }
    /* identify minimum crow flies entry */
    minimumCF=(double)1000000.0;
    for(c=0; c<NumberEndNodes; c++)
{
   if(CFT[c] < minimumCF)
   {
      minimumCFindex=c;
           minimumCF=CFT[c];
   }
}
    /* identify minimum driving entry */
    minimumD=(double)1000000.0;
    for(c=0; c<NumberEndNodes; c++)
{
   if(DT[c] < minimumD)
   {
      minimumDindex=c;
           minimumD=DT[c];
   }
}
    printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",

#if NOT_QUIET

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

    printf("Minimum Driving:    %12s  DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

    printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif

    if(minimumindex == minimumDindex)
{
#if NOT_QUIET

   printf("\n*** CORRECT *** Minimum Euclidian Matches

Minimum Driving Distance\n");
#endif

      GoodCount[i]++;
```

```
Matches Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);


#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif
```

```
fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

    fprintf(goodfile,"Target Node:

fprintf(goodfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
  fprintf(goodfile,"%s  ",

    }
    fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(goodfile,"%s

    }
 fprintf(goodfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(goodfile,"Minimum Euclidian:  %12s



    fprintf(goodfile,"Minimum Driving:    %12s



    fprintf(goodfile,"Minimum Crow Flies: %12s


 }
 else
{

printf("\n*** WRONG *** Minimum Euclidian Does Not
```

```
Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);



%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);


#if NOT_QUIET
```

```
    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
  fprintf(badfile,"%s  ",

    }
    fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%s

    }
 fprintf(badfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(badfile,"Minimum Euclidian:  %12s



    fprintf(badfile,"Minimum Driving:    %12s



    fprintf(badfile,"Minimum Crow Flies: %12s


 }

 if(minimumindex == minimumCFindex)
{
```

```
                                                                    printf("*** CORRECT *** Minimum Euclidian Matches
Minimum Crow Flies Distance\n");
#endif
                                                                }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        break;

        case 7:
        /* Polling Nodes = 7 End Nodes = 4 */
         /* select polling nodes */
         for(p1=0; p1<TotalNodes; p1++)
         {
            for(p2=p1+1; p2<TotalNodes; p2++)
            {
               for(p3=p2+1; p3<TotalNodes; p3++)
               {
                  for(p4=p3+1; p4<TotalNodes; p4++)
                  {
                     for(p5=p4+1; p5<TotalNodes; p5++)
                     {
                        for(p6=p5+1; p6<TotalNodes; p6++)
                        {
                           for(p7=p6+1; p7<TotalNodes; p7++)
                           {
                                 /* we have to check to insure not a target node */
                                if( (p1 != i) && (p2 != i) && (p3 != i) && (p4 != i) && (p5 != i) &&
                                    (p6 != i) && (p7 != i) )
                               {
                                   /* this combination is ok, so store it for later use */
                                    PollingNodeIndexList[0]=p1;
                                    PollingNodeIndexList[1]=p2;
                                    PollingNodeIndexList[2]=p3;
                                    PollingNodeIndexList[3]=p4;
                                    PollingNodeIndexList[4]=p5;
```

```
 PollingNodeIndexList[5]=p6;
 PollingNodeIndexList[6]=p7;
 /* clear selected index */
for(c=0; c<TotalNodes; c++)
{
   Selected[c]=0;
}
 Selected[i] = 1;
Selected[p1] = 1;
Selected[p2] = 1;
Selected[p3] = 1;
Selected[p4] = 1;
Selected[p5] = 1;
Selected[p6] = 1;
Selected[p7] = 1;
/* select end nodes */
 for(e1=0; e1<TotalNodes; e1++)
{
    for(e2=e1+1; e2<TotalNodes; e2++)
   {
      for(e3=e2+1; e3<TotalNodes; e3++)
      {
         for(e4=e3+1; e4<TotalNodes; e4++)
         {
                                  /* this generates all possible combinations of End Nodes
*/
                                  if(!Selected[e1] && !Selected[e2] && !Selected[e3] &&
!Selected[e4] )
                                 {
                                    /* this combination is ok */
                                     combinations++;
                                    EndNodeIndexList[0]=e1;
                                    EndNodeIndexList[1]=e2;
                                    EndNodeIndexList[2]=e3;
                                    EndNodeIndexList[3]=e4;

#if NOT_QUIET
                                      printf("\nTarget Node: %d ",i);
                                 printf("Total Polling Nodes: %d Polling Nodes:
",NumberPollingNodes);
                                      for(c=0; c<NumberPollingNodes; c++)
                                      {
                                         printf("%d ", PollingNodeIndexList[c]+1);
                                      }
                                      printf("Total End Nodes: %d End Nodes:
",NumberEndNodes);
                                      for(c=0; c<NumberEndNodes; c++)
                                      {
```

```
",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);



#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */




#if NOT_QUIET


Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else
```

```
  printf("%d ",EndNodeIndexList[c]+1);
 }
 printf("Target Node: %12s\n",PollingNodes[i+1]);
printf("Total Polling Nodes: %d\nPolling Nodes:

 for(c=0; c<NumberPollingNodes; c++)
 {
  printf("%s  ",

 }
 printf("\nTotal End Nodes: %d\nEnd Nodes:

 for(c=0; c<NumberEndNodes; c++)
 {
  printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
 }
printf("\n");


 result =

         NumberPollingNodes,
       &PollingNodeIndexList[0],
       NumberEndNodes,
       &EndNodeIndexList[0],
       TargetNodeIndex);
 /* calculate crow flys and driving distance between

 for(c=0; c<NumberEndNodes; c++)
 {
  CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
  DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
 }


 for(c=0; c<NumberEndNodes; c++)
 {
    printf("%16s E[%02d] is %12f CrowFlies: %12f

    PollingNodes[EndNodeIndexList[c]+1], c,

 }

 if(!(combinations%100))
{
  /*printf(".");*/
 }
```

```
#endif
                                                                    /* identify minimum euclidian entry */
                                                                    minimum=(double)1000000.0;
                                                                    for(c=0; c<NumberEndNodes; c++)
                                                                  {
                                                                     if(EuclidianDistance[c] < minimum)
                                                                     {
                                                                        minimumindex=c;
                                                                          minimum=EuclidianDistance[c];
                                                                     }
                                                                  }
                                                                    /* identify minimum crow flies entry */
                                                                    minimumCF=(double)1000000.0;
                                                                    for(c=0; c<NumberEndNodes; c++)
                                                               {
                                                                  if(CFT[c] < minimumCF)
                                                                  {
                                                                     minimumCFindex=c;
                                                                          minimumCF=CFT[c];
                                                                  }
                                                               }
                                                                    /* identify minimum driving entry */
                                                                    minimumD=(double)1000000.0;
                                                                    for(c=0; c<NumberEndNodes; c++)
                                                               {
                                                                  if(DT[c] < minimumD)
                                                                  {
                                                                     minimumDindex=c;
                                                                          minimumD=DT[c];
                                                                  }
                                                               }
#if NOT_QUIET
                                                                    printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

                                                                    printf("Minimum Driving:    %12s  DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

                                                                    printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif
                                                                    if(minimumindex == minimumDindex)
                                                               {
#if NOT_QUIET
```

```
Minimum Driving Distance\n");
#endif


Matches Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
```

```
printf("\n*** CORRECT *** Minimum Euclidian Matches


    GoodCount[i]++;

fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

    fprintf(goodfile,"Target Node:

fprintf(goodfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
 fprintf(goodfile,"%s  ",

    }
    fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%s

    }
fprintf(goodfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(goodfile,"Minimum Euclidian:  %12s



    fprintf(goodfile,"Minimum Driving:    %12s



    fprintf(goodfile,"Minimum Crow Flies: %12s


}
else
```

```
#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif


Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
```

```
{
   printf("\n*** WRONG *** Minimum Euclidian Does Not


       BadCount[i]++;

   fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

       fprintf(badfile,"Target Node:

   fprintf(badfile,"Total Polling Nodes: %d\nPolling

       for(c=0; c<NumberPollingNodes; c++)
       {
      fprintf(badfile,"%s  ",

       }
       fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

       for(c=0; c<NumberEndNodes; c++)
       {
      fprintf(badfile,"%s

       }
    fprintf(badfile,"\n");
       for(c=0; c<NumberEndNodes; c++)
       {
      fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

                PollingNodes[EndNodeIndexList[c]+1], c,

       }
       fprintf(badfile,"Minimum Euclidian:  %12s



       fprintf(badfile,"Minimum Driving:    %12s



       fprintf(badfile,"Minimum Crow Flies: %12s
```

```
                                                }

                                                if(minimumindex == minimumCFindex)
                                             {
#if NOT_QUIET
                                               printf("*** CORRECT *** Minimum Euclidian Matches
Minimum Crow Flies Distance\n");
#endif
                                             }
                                 }
                               }
                             }
                           }
                         }
                       }
                     }
                   }
                 }
               }
             }
           }
          }
          break;

          case 8:
          /* Polling Nodes = 8 End Nodes = 3 */
          /* select polling nodes */
          for(p1=0; p1<TotalNodes; p1++)
          {
             for(p2=p1+1; p2<TotalNodes; p2++)
             {
                for(p3=p2+1; p3<TotalNodes; p3++)
                {
                   for(p4=p3+1; p4<TotalNodes; p4++)
                   {
                      for(p5=p4+1; p5<TotalNodes; p5++)
                      {
                         for(p6=p5+1; p6<TotalNodes; p6++)
                         {
                            for(p7=p6+1; p7<TotalNodes; p7++)
                            {
                               for(p8=p7+1; p8<TotalNodes; p8++)
                               {
                                         /* we have to check to insure not a target node */
                                         if( (p1 != i) && (p2 != i) && (p3 != i) && (p4 != i) && (p5 != i) &&
                                              (p6 != i) && (p7 != i) && (p8 != i) )
```

```
{
    /* this combination is ok, so store it for later use */
     PollingNodeIndexList[0]=p1;
     PollingNodeIndexList[1]=p2;
     PollingNodeIndexList[2]=p3;
     PollingNodeIndexList[3]=p4;
     PollingNodeIndexList[4]=p5;
     PollingNodeIndexList[5]=p6;
     PollingNodeIndexList[6]=p7;
     PollingNodeIndexList[7]=p8;
     /* clear selected index */
    for(c=0; c<TotalNodes; c++)
    {
       Selected[c]=0;
    }
     Selected[i] = 1;
    Selected[p1] = 1;
    Selected[p2] = 1;
    Selected[p3] = 1;
    Selected[p4] = 1;
    Selected[p5] = 1;
    Selected[p6] = 1;
    Selected[p7] = 1;
    Selected[p8] = 1;
    /* select end nodes */
     for(e1=0; e1<TotalNodes; e1++)
    {
       for(e2=e1+1; e2<TotalNodes; e2++)
      {
         for(e3=e2+1; e3<TotalNodes; e3++)
         {
                                       /* this generates all possible combinations of End Nodes
*/
                                       if(!Selected[e1] && !Selected[e2] && !Selected[e3] )
                                      {
                                         /* this combination is ok */
                                          combinations++;
                                         EndNodeIndexList[0]=e1;
                                         EndNodeIndexList[1]=e2;
                                         EndNodeIndexList[2]=e3;

#if NOT_QUIET
                                           printf("\nTarget Node: %d ",i);
                                     printf("Total Polling Nodes: %d Polling Nodes:
",NumberPollingNodes);
                                           for(c=0; c<NumberPollingNodes; c++)
                                           {
                                              printf("%d ", PollingNodeIndexList[c]+1);
```

```
",NumberEndNodes);



",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);



#endif

CalculateEuclidianDistance(&EuclidianDistance[0],




target and end nodes */



#if NOT_QUIET


Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);
```

```
 }
 printf("Total End Nodes: %d End Nodes:

 for(c=0; c<NumberEndNodes; c++)
 {
  printf("%d ",EndNodeIndexList[c]+1);
 }
 printf("Target Node: %12s\n",PollingNodes[i+1]);
printf("Total Polling Nodes: %d\nPolling Nodes:

 for(c=0; c<NumberPollingNodes; c++)
 {
  printf("%s  ",

 }
 printf("\nTotal End Nodes: %d\nEnd Nodes:

 for(c=0; c<NumberEndNodes; c++)
 {
  printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
 }
printf("\n");


 result =

         NumberPollingNodes,
       &PollingNodeIndexList[0],
       NumberEndNodes,
       &EndNodeIndexList[0],
       TargetNodeIndex);
 /* calculate crow flys and driving distance between

 for(c=0; c<NumberEndNodes; c++)
 {
  CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
  DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
 }


 for(c=0; c<NumberEndNodes; c++)
 {
    printf("%16s E[%02d] is %12f CrowFlies: %12f

    PollingNodes[EndNodeIndexList[c]+1], c,

 }
```

```
#else
                                                                    if(!(combinations%100))
                                                                   {
                                                                      /*printf(".");*/
                                                                    }
#endif
                                                                    /* identify minimum euclidian entry */
                                                                    minimum=(double)1000000.0;
                                                                    for(c=0; c<NumberEndNodes; c++)
                                                                   {
                                                                      if(EuclidianDistance[c] < minimum)
                                                                      {
                                                                         minimumindex=c;
                                                                           minimum=EuclidianDistance[c];
                                                                      }
                                                                   }
                                                                    /* identify minimum crow flies entry */
                                                                    minimumCF=(double)1000000.0;
                                                                    for(c=0; c<NumberEndNodes; c++)
                                                               {
                                                                   if(CFT[c] < minimumCF)
                                                                  {
                                                                     minimumCFindex=c;
                                                                          minimumCF=CFT[c];
                                                                  }
                                                               }
                                                                    /* identify minimum driving entry */
                                                                    minimumD=(double)1000000.0;
                                                                    for(c=0; c<NumberEndNodes; c++)
                                                               {
                                                                   if(DT[c] < minimumD)
                                                                  {
                                                                     minimumDindex=c;
                                                                          minimumD=DT[c];
                                                                  }
                                                               }
#if NOT_QUIET
                                                                    printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",
PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);
                                                                    printf("Minimum Driving:    %12s  DT[%02d] = %f\n",
PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);
                                                                    printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",
```

```
PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif

#if NOT_QUIET

Minimum Driving Distance\n");
#endif



Matches Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);



PollingNodes[PollingNodeIndexList[c]+1]);


Nodes: ",NumberEndNodes);



",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);
```

```
 if(minimumindex == minimumDindex)
{

printf("\n*** CORRECT *** Minimum Euclidian Matches


    GoodCount[i]++;

fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

    fprintf(goodfile,"Target Node:

fprintf(goodfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
 fprintf(goodfile,"%s  ",

    }
    fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%s

    }
fprintf(goodfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(goodfile,"Minimum Euclidian:  %12s



    fprintf(goodfile,"Minimum Driving:    %12s
```

```
CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);


#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif


Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);


DT[%02d] = %f\n",
```

```
    fprintf(goodfile,"Minimum Crow Flies: %12s


 }
 else
{

printf("\n*** WRONG *** Minimum Euclidian Does Not


    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
  fprintf(badfile,"%s  ",

    }
    fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%s

    }
 fprintf(badfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(badfile,"Minimum Euclidian:  %12s



    fprintf(badfile,"Minimum Driving:    %12s
```

```
PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

                                                                                fprintf(badfile,"Minimum Crow Flies: %12s
CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
                                                                                 }

                                                                                 if(minimumindex == minimumCFindex)
                                                                            {
#if NOT_QUIET
                                                                               printf("*** CORRECT *** Minimum Euclidian Matches

Minimum Crow Flies Distance\n");
#endif
                                                                            }
                                                  }
                                               }
                                            }
                                         }
                                      }
                                   }
                                }
                             }
                          }
                       }
                    }
                 }
              }
              break;

              case 9:
              /* Polling Nodes = 9 End Nodes = 2 */
               /* select polling nodes */
               for(p1=0; p1<TotalNodes; p1++)
               {
                  for(p2=p1+1; p2<TotalNodes; p2++)
                  {
                     for(p3=p2+1; p3<TotalNodes; p3++)
                     {
                        for(p4=p3+1; p4<TotalNodes; p4++)
                        {
                           for(p5=p4+1; p5<TotalNodes; p5++)
                           {
                              for(p6=p5+1; p6<TotalNodes; p6++)
                              {
                                 for(p7=p6+1; p7<TotalNodes; p7++)
```

```
{
     for(p8=p7+1; p8<TotalNodes; p8++)
    {
        for(p9=p8+1; p9<TotalNodes; p9++)
       {
              /* we have to check to insure not a target node */
             if( (p1 != i) && (p2 != i) && (p3 != i) && (p4 != i) && (p5 != i) &&
                  (p6 != i) && (p7 != i) && (p8 != i) && (p9 != i) )
            {
                /* this combination is ok, so store it for later use */
                 PollingNodeIndexList[0]=p1;
                 PollingNodeIndexList[1]=p2;
                 PollingNodeIndexList[2]=p3;
                 PollingNodeIndexList[3]=p4;
                 PollingNodeIndexList[4]=p5;
                 PollingNodeIndexList[5]=p6;
                 PollingNodeIndexList[6]=p7;
                 PollingNodeIndexList[7]=p8;
                 PollingNodeIndexList[8]=p9;
                 /* clear selected index */
                for(c=0; c<TotalNodes; c++)
                {
                   Selected[c]=0;
                }
                 Selected[i] = 1;
                Selected[p1] = 1;
                Selected[p2] = 1;
                Selected[p3] = 1;
                Selected[p4] = 1;
                Selected[p5] = 1;
                Selected[p6] = 1;
                Selected[p7] = 1;
                Selected[p8] = 1;
                Selected[p9] = 1;
                /* select end nodes */
                 for(e1=0; e1<TotalNodes; e1++)
                {
                    for(e2=e1+1; e2<TotalNodes; e2++)
                   {
                                                  /* this generates all possible combinations of End Nodes
*/
                                                  if(!Selected[e1] && !Selected[e2] )
                                                {
                                                   /* this combination is ok */
                                                    combinations++;
                                                  EndNodeIndexList[0]=e1;
                                                  EndNodeIndexList[1]=e2;
```

```
#if NOT_QUIET

",NumberPollingNodes);



",NumberEndNodes);



",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);

",NumberEndNodes);



#endif

CalculateEuclidianDistance(&EuclidianDistance[0],



target and end nodes */



#if NOT_QUIET
```

```
    printf("\nTarget Node: %d ",i);
printf("Total Polling Nodes: %d Polling Nodes:

    for(c=0; c<NumberPollingNodes; c++)
    {
       printf("%d ", PollingNodeIndexList[c]+1);
    }
    printf("Total End Nodes: %d End Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
     printf("%d ",EndNodeIndexList[c]+1);
    }
    printf("Target Node: %12s\n",PollingNodes[i+1]);
  printf("Total Polling Nodes: %d\nPolling Nodes:

    for(c=0; c<NumberPollingNodes; c++)
    {
     printf("%s  ",

    }
    printf("\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
     printf("%s  ",PollingNodes[EndNodeIndexList[c]+1]);
    }
   printf("\n");


    result =

             NumberPollingNodes,
           &PollingNodeIndexList[0],
           NumberEndNodes,
           &EndNodeIndexList[0],
           TargetNodeIndex);
    /* calculate crow flys and driving distance between

    for(c=0; c<NumberEndNodes; c++)
    {
     CFT[c] = CrowFlies[EndNodeIndexList[c]][i];
     DT[c] = DrivingDistance[EndNodeIndexList[c]][i];
    }
```

```
Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);

#else



#endif
```

```
#if NOT_QUIET

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);
```

```
    for(c=0; c<NumberEndNodes; c++)
    {
       printf("%16s E[%02d] is %12f CrowFlies: %12f

       PollingNodes[EndNodeIndexList[c]+1], c,

    }

    if(!(combinations%100))
   {
     /*printf(".");*/
    }

    /* identify minimum euclidian entry */
    minimum=(double)1000000.0;
    for(c=0; c<NumberEndNodes; c++)
   {
      if(EuclidianDistance[c] < minimum)
      {
         minimumindex=c;
           minimum=EuclidianDistance[c];
      }
   }
    /* identify minimum crow flies entry */
    minimumCF=(double)1000000.0;
    for(c=0; c<NumberEndNodes; c++)
{
   if(CFT[c] < minimumCF)
   {
      minimumCFindex=c;
           minimumCF=CFT[c];
   }
}
    /* identify minimum driving entry */
    minimumD=(double)1000000.0;
    for(c=0; c<NumberEndNodes; c++)
{
   if(DT[c] < minimumD)
   {
      minimumDindex=c;
           minimumD=DT[c];
   }
}

    printf("Minimum Euclidian:  %12s   E[%02d] = %f\n",
```

```
PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);
#endif

#if NOT_QUIET

Minimum Driving Distance\n");
#endif


Matches Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


Nodes: ",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);
```

```
 printf("Minimum Driving:    %12s  DT[%02d] = %f\n",


 printf("Minimum Crow Flies: %12s CFT[%02d] = %f\n",


 if(minimumindex == minimumDindex)
{
printf("\n*** CORRECT *** Minimum Euclidian Matches


    GoodCount[i]++;

fprintf(goodfile,"*** CORRECT *** Minimum Euclidian

    fprintf(goodfile,"Target Node:

fprintf(goodfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
 fprintf(goodfile,"%s  ",

    }
    fprintf(goodfile,"\nTotal End Nodes: %d\nEnd

    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%s

    }
fprintf(goodfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
 fprintf(goodfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(goodfile,"Minimum Euclidian:  %12s
```

```
DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);


CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);


#if NOT_QUIET

Matcs Minimum Driving Distance\n");
#endif


Not Matcs Minimum Driving Distance\n");

%12s\n",PollingNodes[i+1]);

Nodes: ",NumberPollingNodes);


PollingNodes[PollingNodeIndexList[c]+1]);


",NumberEndNodes);


",PollingNodes[EndNodeIndexList[c]+1]);




%12f Driving: %12f\n",

EuclidianDistance[c],CFT[c],DT[c]);


E[%02d] = %f\n",
```

```
    fprintf(goodfile,"Minimum Driving:    %12s


    fprintf(goodfile,"Minimum Crow Flies: %12s


 }
 else
{

printf("\n*** WRONG *** Minimum Euclidian Does Not


    BadCount[i]++;

fprintf(badfile,"*** WRONG *** Minimum Euclidian Does

    fprintf(badfile,"Target Node:

fprintf(badfile,"Total Polling Nodes: %d\nPolling

    for(c=0; c<NumberPollingNodes; c++)
    {
  fprintf(badfile,"%s  ",

    }
    fprintf(badfile,"\nTotal End Nodes: %d\nEnd Nodes:

    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%s

    }
 fprintf(badfile,"\n");
    for(c=0; c<NumberEndNodes; c++)
    {
  fprintf(badfile,"%16s E[%02d] is %12f CrowFlies:

            PollingNodes[EndNodeIndexList[c]+1], c,

    }
    fprintf(badfile,"Minimum Euclidian:  %12s
```

```
PollingNodes[EndNodeIndexList[minimumindex]+1],minimumindex,minimum);

DT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumDindex]+1],minimumDindex,minimumD);

CFT[%02d] = %f\n",

PollingNodes[EndNodeIndexList[minimumCFindex]+1],minimumCFindex,minimumCF);



#if NOT_QUIET

Minimum Crow Flies Distance\n");
#endif
                                                  }
                                               }
                                            }
                                         }
                                      }
                                   }
                                }
                             }
                          }
                       }
                    }
                 }
              }
              break;


              default:
                 printf("Major Error default case\n");
                  exit(0);
                  break;
        }
     }
     printf("Closing %s file...\n",BadFilename);
     fclose(badfile);
     printf("Closing %s file...\n", GoodFilename);
     fclose(goodfile);
```

```
                 fprintf(badfile,"Minimum Driving:    %12s

                 fprintf(badfile,"Minimum Crow Flies: %12s


  }

  if(minimumindex == minimumCFindex)
{
   printf("*** CORRECT *** Minimum Euclidian Matches

}
```

```
      printf("Finished Polling Nodes: %d End Nodes: %d\n", NumberPollingNodes, NumberEndNodes);
      printf("\n");
      for(i=0; i<13; i++)
      {
         printf("Target node: %2d %12s Good: %6lu Bad: %6lu\n",i,PollingNodes[i+1],GoodCount[i],BadCount[i]);
      }
      printf("Total Combinations: %lu\n", combinations);

   }

   printf("Version 03 March 2007\n");

   printf("Closing out.csv file...\n");
   fclose(outfile);

   return 0;
}

/* end CollectIt12.c */
```

Appendix B: CollectIt12.c “C” Program

```
/* begin Area12.c */

/* A program to analyze IP Geolocation area data */
/* For 12 Total Nodes */

/* includes */
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/* defines */
#define DEBUG_MODE 0
#define NOT_QUIET 0
#define TOTAL_NODES ((int) 12)
#define MAX_RADIUS ((int) 4000)
#define PI ((double)3.14159265358979323846264338327950288419716939937510)
#define TWOPIBY360 ( (PI*(double)2.0)/((double)360))
#define XMAX ((int)11938)
#define YMAX ((int)9469)

/* structures */
struct ACircleQuad
{
   int XRef;
   int YRef;
   int Radius;
   int XMinB;
   int XMaxB;
   int YMinB;
   int YMaxB;
   int Length[MAX_RADIUS];
};

/* function prototypes */
void ClearCircles(void);
void DefineCircle(int WhichCircle, int XRef, int YRef, int Radius);
int IsInCircle(int WhichCircle, int X, int Y);
unsigned long CalcArea(void);
void CalculateDistance(int X, int Y);
int FindMinumum(int X, int Y);
```

```
/* globals */
struct ACircleQuad Circles[TOTAL_NODES];
double Delays[TOTAL_NODES][TOTAL_NODES];
double SOL[TOTAL_NODES][TOTAL_NODES];
double CB[TOTAL_NODES][TOTAL_NODES];
double LOC[TOTAL_NODES][2];
int ILOC[TOTAL_NODES][TOTAL_NODES];
char PollingNodes[TOTAL_NODES+1][100];
int Included[TOTAL_NODES];
unsigned int Distance[TOTAL_NODES];

/* functions */
void ClearCircles(void)
{
   int Circle, i;

#if DEBUG_MODE
   printf("ClearCircles\n");
#endif

   for(Circle=0; Circle<TOTAL_NODES; Circle++)
   {
      Circles[Circle].XRef=(int)0;
      Circles[Circle].YRef=(int)0;
      Circles[Circle].Radius=(int)0;
      Circles[Circle].XMinB=(int)0;
      Circles[Circle].XMaxB=(int)0;
      Circles[Circle].YMinB=(int)0;
      Circles[Circle].YMaxB=(int)0;
      for(i=0; i<MAX_RADIUS; i++)
      {
        Circles[Circle].Length[i]=(int)0;
      }
   }
}

void DefineCircle(int WhichCircle, int XRef, int YRef, int Radius)
{
   int Angle, x, y, i;

#if DEBUG_MODE
   printf("DefineCircle Circle: %d XRef: %d YRef: %d Radius: %d\n", WhichCircle, XRef, YRef, Radius);
#endif

   Circles[WhichCircle].XRef=XRef;
   Circles[WhichCircle].YRef=YRef;
   Circles[WhichCircle].Radius=Radius;
```

```
   /* do this calc here because we need it later for speed */
   Circles[WhichCircle].XMinB = XRef - Radius;
   Circles[WhichCircle].XMaxB = XRef + Radius;
   Circles[WhichCircle].YMinB = YRef - Radius;
   Circles[WhichCircle].YMaxB = YRef + Radius;

   /* clear out old lengths */
   for(i=0; i<MAX_RADIUS; i++)
   {
      Circles[WhichCircle].Length[i]=(int)0;
   }

   for(Angle=0; Angle<=90; Angle++)
   {
      y = (int) ( (double)Radius * sin(TWOPIBY360*(double)Angle) );
      x = (int) ( (double)Radius * cos(TWOPIBY360*(double)Angle) );
      for(i=0; (i<=y)&&(i<MAX_RADIUS); i++)
      {
         if(x > Circles[WhichCircle].Length[i])
        {
           Circles[WhichCircle].Length[i] = x;
        }
      }
   }
#if DEBUG_MODE
   for(i=0; i<4000; i++)
   {
      printf("Circle: %d i: %d Length: %d\n", WhichCircle, i, Circles[WhichCircle].Length[i]);
   }
#endif
}

int IsInCircle(int WhichCircle, int X, int Y)
{
   int XMinB, XMaxB, YMinB, YMaxB, XDelta, YDelta;

   /* quick check to see if the location is outside the box */
   if( (X < Circles[WhichCircle].XMinB) ||
       (X > Circles[WhichCircle].XMaxB) ||
       (Y < Circles[WhichCircle].YMinB) ||
       (Y > Circles[WhichCircle].YMaxB) )
   {
      /* it is not in the square, so no more checking is needed */
#if DEBUG_MODE
      printf("IsInCircle? NO Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
      return(0);
```

```
   }
   XDelta = X - Circles[WhichCircle].XRef;
   YDelta = Y - Circles[WhichCircle].YRef;
   if( (XDelta >= 0) && (YDelta >= 0) )
   {
      /* we are in Quadrant I */
      if(XDelta <= Circles[WhichCircle].Length[YDelta])
      {
#if DEBUG_MODE
         printf("IsInCircle? YES Quadrant I Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(1);
      }
      else
      {
#if DEBUG_MODE
         printf("IsInCircle? NO Quadrant I Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(0);
      }
   }

   if( (XDelta < 0) && (YDelta >= 0) )
   {
      /* we are in Quadrant II */
      if(XDelta >= (-1 * Circles[WhichCircle].Length[YDelta]) )
      {
#if DEBUG_MODE
         printf("IsInCircle? YES Quadrant II Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(1);
      }
      else
      {
#if DEBUG_MODE
         printf("IsInCircle? NO Quadrant II Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(0);
      }
   }

   if( (XDelta < 0) && (YDelta < 0) )
   {
      /* we are in Quadrant III */
      if(XDelta >= (-1 * Circles[WhichCircle].Length[-1 * YDelta]) )
      {
#if DEBUG_MODE
```

```
         printf("IsInCircle? YES Quadrant III Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(1);
      }
      else
      {
#if DEBUG_MODE
         printf("IsInCircle? NO Quadrant III Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(0);
      }
   }

   if( (XDelta >= 0) && (YDelta < 0) )
   {
      /* we are in Quadrant IV */
      if(XDelta <= Circles[WhichCircle].Length[-1 * YDelta])
      {
#if DEBUG_MODE
         printf("IsInCircle? YES Quadrant IV Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(1);
      }
      else
      {
#if DEBUG_MODE
         printf("IsInCircle? NO Quadrant IV Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
#endif
         return(0);
      }
   }
   /* should never get here */
   printf("IsInCircle? FAILURE Circle: %d X: %d Y: %d\n", WhichCircle, X, Y);
   exit(0);
}

unsigned long CalcArea(void)
{
   int i, j, flag;
   unsigned long area;

   for(i=0; i<XMAX; i++)
   {
      for(j=0; j<YMAX; j++)
      {
         /* walk the whole map */
        /* the first negative should skip the rest */
```

```
flag=1;

 if(Included[0])
{
   if(!IsInCircle(0, i, j))
   {
      flag=0;
      goto bottom;
   }
 }
 if(Included[1])
{
   if(!IsInCircle(1, i, j))
   {
      flag=0;
      goto bottom;
   }
 }
 if(Included[2])
{
   if(!IsInCircle(2, i, j))
   {
      flag=0;
      goto bottom;
   }
 }
 if(Included[3])
{
   if(!IsInCircle(3, i, j))
   {
      flag=0;
      goto bottom;
   }
 }
 if(Included[4])
{
   if(!IsInCircle(4, i, j))
   {
      flag=0;
      goto bottom;
   }
 }
 if(Included[5])
{
   if(!IsInCircle(5, i, j))
   {
      flag=0;
```

```
      goto bottom;
   }
}
if(Included[6])
{
   if(!IsInCircle(6, i, j))
   {
      flag=0;
      goto bottom;
   }
}
if(Included[7])
{
   if(!IsInCircle(7, i, j))
   {
      flag=0;
      goto bottom;
   }
}
if(Included[8])
{
   if(!IsInCircle(8, i, j))
   {
      flag=0;
      goto bottom;
   }
}
if(Included[9])
{
   if(!IsInCircle(9, i, j))
   {
      flag=0;
      goto bottom;
   }
}
if(Included[10])
{
   if(!IsInCircle(10, i, j))
   {
      flag=0;
      goto bottom;
   }
}
if(Included[11])
{
   if(!IsInCircle(11, i, j))
   {
```

```
               flag=0;
               goto bottom;
            }
         }

      bottom:
        if(flag)
        {
           area++;
        }

      }
   }
   return(area);
}

void CalculateDistance(int X, int Y)
{
   int i, j;

   for(i=0; i<13; i++)
   {
      Distance[i] = (unsigned long)0;
   }

   for(i=0; i<13; i++)
   {
     Distance[i] = (unsigned long) sqrt( (double)( ( (unsigned long)(X - ILOC[i][0]) * (unsigned long)(X - ILOC[i][0]) ) +
                                                   ( (unsigned long)(Y - ILOC[i][1]) * (unsigned long)(Y - ILOC[i][1]) ) ) );
   }
}

int FindMinumum(int X, int Y)
{
   int i, j;
   int MinIndex;
   unsigned long MinVal;

   MinIndex = 0;
   MinVal = 10000000;
   for(i=0; i<TOTAL_NODES; i++)
   {
#if 0
      /* exclude node if X and Y match */
      if( (X == ILOC[i][0]) && (Y == ILOC[i][1]) )
      {
         /* this is a match, so don't allow it to be selected */
```

```
      }
      else
      {
      }
#endif
      if(Distance[i] < MinVal)
      {
         MinIndex = i;
        MinVal = Distance[i];
      }
   }
   return(MinIndex);
}


/* main function */

int main(int argc, char* argv[])
{
   char InputLine[5000];
   char Token[20][100];
   char PN[TOTAL_NODES+1];
   char TN[TOTAL_NODES+1];
   char Selected[13];
   char GoodFilename[100];
   char BadFilename[100];

   unsigned long combinations;
   unsigned long GoodCount[TOTAL_NODES];
   unsigned long BadCount[TOTAL_NODES];
   unsigned long Area;
   unsigned long TTLHArea[TOTAL_NODES];

   int result;
   int i;
   int j;
   int c;
   int Length;
   int Tokens;
   int LineCount;
   int First;
   int TotalNodes;
   int NumberPollingNodes;
   int CurrentPollingNodes;
   int PollingNodeIndexList[TOTAL_NODES];
   int NumberEndNodes;
   int EndNodeIndexList[TOTAL_NODES];
```

```
int TargetNodeIndex;
int minimumindex;
int minimumCFindex;
int minimumDindex;
int done;
int MinimumIndex;
int CurNode;
int minx, maxx, miny, maxy;

double CrowFlies[TOTAL_NODES][TOTAL_NODES];
double DrivingDistance[TOTAL_NODES][TOTAL_NODES];
double EuclidianDistance[TOTAL_NODES];
double CFT[TOTAL_NODES];
double DT[TOTAL_NODES];
double minimum;
double minimumCF;
double minimumD;

FILE *goodfile;
FILE *badfile;
FILE *outfile;
FILE *costfile;

printf("Version 03 March 2007\n");

printf("Opening out.csv for write...\n");
/*costfile=fopen("d:\\out.csv","w");*/
outfile=fopen("./out.csv","w");
if(outfile==NULL)
{
   printf("\n Error cannot open file out.csv\a ");
   exit(0);
}

printf("Reading DelayData.csv file...\n");
/*costfile=fopen("d:\\DelayData.csv","r");*/
costfile=fopen("./DelayData.csv","r");
if(costfile==NULL)
{
   printf("\n Error cannot open file DelayData.csv\a ");
   exit(0);
}
LineCount = 0;
First = 1;
while(fgets(InputLine,4999,costfile) != NULL)
{
   /* get a line */
```

```
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* delimiter */
             Token[Tokens][j] = 0;
            j = 0;
            Tokens++;
         }
         else
         {
            /* character */
             Token[Tokens][j++] = InputLine[i];
         }
      }
      Token[Tokens++][j] = 0;
      /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
      if(First)
      {
         for(i=0; i<Tokens; i++)
         {
            strcpy(PollingNodes[i],&Token[i][0]);
             /*sprintf(PollingNodes[i], "%s", Token[i][0]);*/
         }
         First=0;
      }
      else
      {
         /* data */
         for(i=0; i<Tokens; i++)
         {
            if(i)
            {
               Delays[LineCount-2][i-1] = (double)atof(Token[i]);
            }
            /*sprintf(PollingNodes[i], "%s", Token[i][0]);*/
         }
      }
#if DEBUG_MODE
      printf("Line %d with %d tokens\n",LineCount,Tokens);
      for(i=0; i<Tokens; i++)
      {
```

```
         printf("Token %d is %s\n",i,&Token[i][0]);
      }
#endif
      InputLine[Length-1]=0;
      fprintf(outfile, "%s\n", InputLine);
   }
   printf("Closing DelayData.csv file...\n");
   fclose(costfile);

   printf("Reading SOL.csv file...\n");
   /*costfile=fopen("d:\\SOL.csv","r");*/
   costfile=fopen("./SOL.csv","r");
   if(costfile==NULL)
   {
      printf("\n Error cannot open file SOL.csv\a ");
      exit(0);
   }
   LineCount = 0;
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* delimiter */
            Token[Tokens][j] = 0;
            j = 0;
            Tokens++;
         }
         else
         {
            /* character */
            Token[Tokens][j++] = InputLine[i];
         }
      }
      Token[Tokens++][j] = 0;
      /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
      if(First)
      {
         First=0;
```

```
      }
      else
      {
         /* data */
         for(i=0; i<Tokens; i++)
         {
            if(i)
            {
               SOL[LineCount-2][i-1] = (double)atof(Token[i]);
            }
         }
      }
#if DEBUG_MODE
      printf("Line %d with %d tokens\n",LineCount,Tokens);
      for(i=0; i<Tokens; i++)
      {
         printf("Token %d is %s\n",i,&Token[i][0]);
      }
#endif
      InputLine[Length-1]=0;
      fprintf(outfile, "%s\n", InputLine);
   }
   printf("Closing SOL.csv file...\n");
   fclose(costfile);

   printf("Reading CB.csv file...\n");
   /*costfile=fopen("d:\\CB.csv","r");*/
   costfile=fopen("./CB.csv","r");
   if(costfile==NULL)
   {
      printf("\n Error cannot open file CB.csv\a ");
      exit(0);
   }
   LineCount = 0;
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
```

```
            /* delimiter */
             Token[Tokens][j] = 0;
            j = 0;
            Tokens++;
         }
         else
         {
            /* character */
             Token[Tokens][j++] = InputLine[i];
         }
      }
      Token[Tokens++][j] = 0;
      /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
      if(First)
      {
        First=0;
      }
      else
      {
         /* data */
         for(i=0; i<Tokens; i++)
         {
            if(i)
            {
               CB[LineCount-2][i-1] = (double)atof(Token[i]);
            }
         }
      }
#if DEBUG_MODE
      printf("Line %d with %d tokens\n",LineCount,Tokens);
      for(i=0; i<Tokens; i++)
      {
         printf("Token %d is %s\n",i,&Token[i][0]);
      }
#endif
      InputLine[Length-1]=0;
      fprintf(outfile, "%s\n", InputLine);
   }
   printf("Closing CB.csv file...\n");
   fclose(costfile);

   printf("Reading LOC.csv file...\n");
   /*costfile=fopen("d:\\LOC.csv","r");*/
   costfile=fopen("./LOC.csv","r");
   if(costfile==NULL)
   {
      printf("\n Error cannot open file LOC.csv\a ");
```

```
      exit(0);
   }
   LineCount = 0;
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* delimiter */
            Token[Tokens][j] = 0;
            j = 0;
            Tokens++;
         }
         else
         {
            /* character */
            Token[Tokens][j++] = InputLine[i];
         }
      }
      Token[Tokens++][j] = 0;
      /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
      if(First)
      {
        First=0;
      }
      else
      {
         /* data */
        LOC[LineCount-2][0] = (double)atof(Token[1]);
        LOC[LineCount-2][1] = (double)atof(Token[2]);
         ILOC[LineCount-2][0] = (int)LOC[LineCount-2][0];
         ILOC[LineCount-2][1] = (int)LOC[LineCount-2][1];
      }
#if DEBUG_MODE
      printf("Line %d with %d tokens\n",LineCount,Tokens);
      for(i=0; i<Tokens; i++)
      {
         printf("Token %d is %s\n",i,&Token[i][0]);
      }
```

```
#endif
      InputLine[Length-1]=0;
      fprintf(outfile, "%s\n", InputLine);
   }
   printf("Closing LOC.csv file...\n");
   fclose(costfile);

   printf("Reading CrowFlies.csv file...\n");
   /*costfile=fopen("d:\\CrowFlies.csv","r");*/
   costfile=fopen("./CrowFlies.csv","r");
   if(costfile==NULL)
   {
      printf("\n Error cannot open file CrowFlies.csv\a ");
      exit(0);
   }
   LineCount = 0;
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* comma delimiter */
            Token[Tokens][j] = 0;
            j = 0;
            Tokens++;
         }
         else
         {
            /* character */
            Token[Tokens][j++] = InputLine[i];
         }
      }
      Token[Tokens++][j] = 0;
      /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
      if(First)
      {
         First=0;
      }
      else
```

```
      {
         /* data */
         for(i=0; i<Tokens; i++)
         {
             if(i)
             {
                CrowFlies[LineCount-2][i-1] = (double)atof(Token[i]);
             }
         }
       }
#if DEBUG_MODE
       printf("Line %d with %d tokens\n",LineCount,Tokens);
       for(i=0; i<Tokens; i++)
       {
          printf("Token %d is %s\n",i,&Token[i][0]);
       }
#endif
       InputLine[Length-1]=0;
   }
   printf("Closing CrowFlies.csv file...\n");
   fclose(costfile);

   printf("Reading Driving.csv file...\n");
   /*costfile=fopen("d:\\Driving.csv","r");*/
   costfile=fopen("./Driving.csv","r");
   if(costfile==NULL)
   {
      printf("\n Error cannot open file Driving.csv\a ");
      exit(0);
   }
   LineCount = 0;
   First = 1;
   while(fgets(InputLine,4999,costfile) != NULL)
   {
      /* get a line */
      LineCount++;
      Length = strlen(InputLine);
      /* tokenize it */
      Tokens=0;
      j=0;
      for(i=0; i<Length; i++)
      {
         if(InputLine[i] == 44)
         {
            /* comma delimiter */
             Token[Tokens][j] = 0;
            j = 0;
```

```
            Tokens++;
          }
          else
          {
             /* character */
              Token[Tokens][j++] = InputLine[i];
          }
       }
       Token[Tokens++][j] = 0;
       /*printf("Line %d has %d Tokens\n",LineCount,Tokens);*/
       if(First)
       {
          First=0;
       }
       else
       {
          /* data */
          for(i=0; i<Tokens; i++)
          {
             if(i)
             {
                DrivingDistance[LineCount-2][i-1] = (double)atof(Token[i]);
             }
          }
       }
#if DEBUG_MODE
       printf("Line %d with %d tokens\n",LineCount,Tokens);
       for(i=0; i<Tokens; i++)
       {
          printf("Token %d is %s\n",i,&Token[i][0]);
       }
#endif
       InputLine[Length-1]=0;
   }
   printf("Closing Driving.csv file...\n");
   fclose(costfile);

   /* replace end of line carrige return with null termination for Tampa */
   PollingNodes[TOTAL_NODES][5]=0;

   /* generate combos */
   TotalNodes=TOTAL_NODES;

#if DEBUG_MODE
   for(i=0; i<TOTAL_NODES+1; i++)
   {
      printf("Polling Node %d is %s\n",i,PollingNodes[i]);
```

```
   }
   for(i=0; i<TOTAL_NODES; i++)
   {
      for(j=0; j<TOTAL_NODES; j++)
      {
         printf("Row: %d Col: %d CrowFlies: %f Driving: %f Delay: %f SOL: %f CB: %f\n", i,j,CrowFlies[i][j],
                DrivingDistance[i][j],Delays[i][j],SOL[i][j], CB[i][j]);
      }
   }
   for(i=0; i<TOTAL_NODES; i++)
   {
     printf("LOC i: %d X: %f Y: %f IX: %d IY: %d\n", i,LOC[i][0],LOC[i][1],ILOC[i][0],ILOC[i][1]);
   }
#endif

   printf("Calculating Area of 11 Overlapping Nodes using SOL for each Target Node\n");
   for(i=0; i<TOTAL_NODES; i++)
   {
      /* pick a target node */
      ClearCircles();
      /* set all included, then exclude target */
      for(j=0; j<TOTAL_NODES; j++)
      {
         Included[j]=1;
      }
      Included[i]=0;
      /* create circles */
      for(j=0; j<TOTAL_NODES; j++)
      {
         if(Included[j])
        {
            /* create each of the circles except the target node */
            DefineCircle((int)j, ILOC[j][0], ILOC[j][1], (int)SOL[i][j] );
         }
      }
      Area = CalcArea();
      printf("Target: %2d  Name: %14s  Area: %10lu\n", i, PollingNodes[i+1], Area);
   }

   printf("Calculating Area of 11 Overlapping Nodes using CB for each Target Node\n");
   for(i=0; i<TOTAL_NODES; i++)
   {
      /* pick a target node */
      ClearCircles();
      /* set all included, then exclude target */
      for(j=0; j<TOTAL_NODES; j++)
      {
```

```
         Included[j]=1;
      }
      Included[i]=0;
      /* create circles */
      for(j=0; j<TOTAL_NODES; j++)
      {
         if(Included[j])
        {
            /* create each of the circles except the target node */
            DefineCircle((int)j, ILOC[j][0], ILOC[j][1], (int)CB[i][j] );
         }
      }
      Area = CalcArea();
      printf("Target: %2d  Name: %14s  Area: %10lu\n", i, PollingNodes[i+1], Area);
   }


   minx=20000;
   maxx=0;
   miny=20000;
   maxy=0;
   for(i=0; i<TOTAL_NODES; i++)
   {
     if(ILOC[i][0] < minx)
     {
        minx = ILOC[i][0];
     }
     if(ILOC[i][0] > maxx)
     {
        maxx = ILOC[i][0];
     }
     if(ILOC[i][1] < miny)
     {
        miny = ILOC[i][1];
     }
     if(ILOC[i][1] > maxy)
     {
        maxy = ILOC[i][1];
     }
   }
#if 0
   printf("Min X: %d Max X: %d Min Y: %d Max Y: %d\n",minx,maxx,miny,maxy);

   /* This works but takes too long to run due to floating point calculations */
   printf("Calculating TTLH Area surrounding each of the 12 Nodes\n");
   for(i=0; i<TOTAL_NODES; i++)
   {
```

```
      TTLHArea[i]=0;
   }
   for(i=0; i<XMAX; i++)
   {
      if(!(i%10))
      {
         printf(".");
      }

      for(j=0; j<YMAX; j++)
      {
         /* walk the whole map */
         CalculateDistance(i, j);
         MinimumIndex = FindMinumum(i, j);
         TTLHArea[MinimumIndex]++;
      }
   }
   for(i=0; i<TOTAL_NODES; i++)
   {
      printf("\nNode: %2d  Name: %14s  Area: %10lu\n", i, PollingNodes[i+1], TTLHArea[i]);
   }
#endif

   printf("Version 03 March 2007\n");

   printf("Closing out.csv file...\n");
   fclose(outfile);

   return 0;
}

/* end Area12.c */
```

Bibliography

[1] Anderson M., Bansal A, Doctor B, Hadjiyiannis G, C. Herringshaw, E. Karplus, and D. Muniz, “Method and apparatus for estimating a geographic location of a networked entity,” United States Patent 6,684,250, Filed 3 April 2001. Issued 27 January 2004.

[2] Ballintijin, Gerco and Maarten van Stten. 2000. "Characterizing Internet Performance to Support Wide-area Application development." *Operating Systems Review*.

[3] Clarson, John R. 2005. *Geolocation of a Node on a Local Area Network.* Dayton OH: Air Force Institute of Technology.

[4] Davis C., P. Vixie, T. Goodwin and I. Dickinson. 1996. "A means for expressing location information in the domain name system." *RFC 1876*.

[5] Dingledine R., Mathewson N., Syverson P., *“Tor: the second-generation onion router”* Proceeding of the 13th USENIX Security Symposium, August 2004, pp. 303-320

[6] Gueye, Bamba, Steve Uhlig, Artur Ziviani and Serge Fdida. 2006. "Leveraging Buffering Delay Estimation for Geolocation of Internet Hosts." *IFIP Networking 2006*(3976):319.

[7] Gueye, Bamba, Artur Ziviani, Mark Crovella and Serge Fdida. 2004. "Constraint-Based Geolocation of Internet hosts." *ACM/SIGCOMM Internet Measurement Conference*:288.

[8] Huffman, Stephen M. and Michael H. Reifer. 2005. "Method for geolocating logical network addresses." 752898(6,947,978).

[9] IEEE Std 802-2001. 2002. *IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture.*

[10] Kish, SA. 1999. "Betting on the Net: An Analysis of the Governments Role in Addressing INternet Gambling." *Federal Communications Law Journal*.

[11] Microsoft Corp. 2006. "Microsoft Windows XP - Ping.", Retrieved December 13, 2006 (http://www.microsoft.com/resources/documentation/windows).

[12] Muir, J., & van Oorschot, P. C. (2006). *Internet geolocation and evasion*Carleton University.

[13] Navas, Julio and Tomasz Imielinski. 1997. "GeoCast- Geographic Addressing and Routing." *3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*:66-66-76.

[14] OPNET Modeler 12.0 Product Documentation -Standard Models User Guide 16 RIP Model User Guide, OPNET Technologies, 2006.

[15] Padmanabhan, Venka and Lakshminarayanan Subramanian. 2001. "An Investigation of Geographic Mapping Techniques for Internet Hosts." *SIGCOMM*(1-58113-411-8/01/008):173.

[16] Parekh S, Friedman R., Tibrewala, and Lutch B, “Systems and methods for determining collecting and using geographic locations of Internet users,” United States Patent 6,757,740, Filed 31 March 2000, Issued 29 June 2004.

[17] Percacci, R. and A. Vespignani. 2003. "Scale-Free behavior of the Internet global performance." *The European Physical Journal* 411-414.

[18] Peterson, Larry L. and Bruce S. Davie. 2003. *Computer Networks, A Systems Approach.* San Fransico CA: Elsevier Science.

[19] PING(8) FreeBSD System Manager’s Manual, www.kerneled.com/doc/man/freebsd/man8/ping.html, 2002.

[20] Quova Inc. 2006 “GeoPoint by Quova: IP Intelligence and Internet Geolocation”

[21] Sorgaard, D. (2004). *Geographic location of a computer node examining a time-to0location algorithm and multiple autonomous system networks.* Unpublished Masters, Air Force Institute of Technology.

[22] Subramanian, Venkata, and Katz. 2002 “Geographic Properties of Internet Routing” *UNSENIX Annual Technical Conference*

[23] Stallings, William. 2004. *Data and Computer Communications.* Upper Saddle River NJ: Pearson Education.

[24] Subramanian, Lakshminarayanan, Venka Padmanabhan and Randy Katz. 2002. "Geographic Properties of Internet Routing." *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*

[25] Tanenbaum, Andrew S. 1999. *Computer Networks.* 3rd ed. Upper Saddle River NJ: Prentice Hall.

[26] Turnbaugh, Eugene. 2004. *Geographically Locating an Internet Node using Network Latency Measurement.* Dayton OH: Air Force Institute of Technology.

[32]Traceroute, Sun Operating System version 5.8, Maintenance Commands, 1999.

[28] United States Geological Survey *"Science in your State: Texas"* retrieved on February 26 2007. http://www.usgs.gov/

[29] Visual Route, "Visualroute traceroute server: Trace IP address, trace route, IP trace, IP address locations" Retrieved on February 27 2007, (http://visualroute.visualware.com/)

[30] Wynne, M., & Moseley, M. (2005**).** *SECAF/CSAF letter to airmen: Mission statement*

[31] Wynne, Michael W, Secretary of the Air Force. “Cyberspace as a Domain in which the Air Force Flies and Fights” Address to the C4ISR Integration Conference, Cyrstal City VA, November 2, 2006

[32] Ziviani, Artur, Serge Fdida, Jose Rezende and Duarte, Otto Carlos M.B. 2005. "Improving the accuracy of measurement-based geographic location of Internet hosts." *Computer Networks* 503-523(47):503.

REPORT DOCUMENTATION PAGE

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**
22-03-2007	**Master's Thesis**

4. TITLE AND SUBTITLE

Internet Protocol Geolocation: Development of a Delay-Based Hybrid Methodology for Locating the Geographic Location of a Network Node

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Roehl John M., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Internet Protocol Geolocation (IP Geolocation), the process of determining the approximate geographic location of an IP addressable node, has proven useful in a wide variety of commercial applications. Commercial applications of IP Geolocation include market research, redirection for performance enhancement, restricting content, and combating fraud. The potential for military applications include securing remote access via geographic authentication, intelligence collection, and cyber attack attribution.

IP Geolocation methods can be divided into three basic categories based upon which information is used to determine the geographic location of the given IP address: 1) Information contained in databases, 2) information that is leaked during connections with the IP of interest, and 3) network-based routing and timing information. This thesis focused upon an analysis in the third category: delay-based methods for IP Geolocation. Specifically, a comparative analysis of three existing delay-based IP Geolocation methods: Upper-bound Multilateration (UBM), Constraint Based Geolocation (CBG), and Time to Location Heuristic (TTLH) is conducted using a simulated network. Based upon analysis of the results, a new hybrid methodology is proposed to improve the accuracy when conducting IP Geolocation. Simulations of the new hybrid methodology show that the hybrid methodology is superior to all existing delay-based methods for IP Geolocation.

15. SUBJECT TERMS

Internet, Location, Triangulation,

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
REPORT	**ABSTRACT**	**c. THIS PAGE**			Michael R. Grimaila, PhD (ENV)
U	**U**	**U**	**UU**	185	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636, ext 4800; e-mail: Michael.Grimaila@afit.edu